MASTER EN SCIENCES INFORMATIQUES

Analyse dynamique de fichiers audit-trail pour la détection de virus informatiques

Delhez, Didier; Hock, Michael

Award date:
1996

Awarding institution:
Universite de Namur

FACULTÉS
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

Analyse dynamique de fichiers audit-trail pour la détection de virus informatiques

Didier DELHEZ
Michael HOCK

Professeur : Baudouin Le Charlier

Mémoire présenté en vue
de l'obtention du grade de
Licencié et Maître en Informatique

Année académique 1995 - 1996

RUE GRANDGAGNAGE, 21 , B - 5000 NAMUR (BELGIUM)

Résumé

Mots clés : virus, audit-trail, A.S.A.X., modélisation de stratégie d'infection

Les virus informatiques constituent un problème permanent dans le souci de sécurité des gens pour leurs ordinateurs. Ces programmes parasites sont de plus en plus perfectionnés et les outils pour les détecter sont de plus en plus difficiles à mettre à jour.
Face à ce problème de mise à jour, des chercheurs ont imaginé un autre type d'analyse qui ne soit plus statique mais dynamique. Ces analyses dynamiques peuvent être implémentées grâce au système ASAX permettant l'analyse de fichiers audit-trail générés à partir de l'exécution de virus sous une émulation PC-DOS. Le développement de programmes d'analyse est facilité par la représentation conceptuelle via des diagrammes modélisant les stratégies d'infection des virus, stratégies beaucoup moins nombreuses que les virus eux-mêmes.

Abstract

Keywords : viruses, audit trails, A.S.A.X., infection strategy modelling

Computer viruses constitute a permanent security problem for computer users. These parasitic programs are more and more sophisticated and the tools for their detection are more and more hard to update.
In order to face this updating problem, researchers have imagined another type of analysis, which is not static anymore, but dynamic. These dynamic analysis can be implemented thanks to ASAX, which allows to analyse audit trails generated by the virus execution in a PC-DOS emulation. The development of analysis programs is simplified through a conceptual representation using diagrams modelling virus infection strategies, which are less numerous than viruses themselves.

Nous souhaiterions remercier ici :

Monsieur Baudouin Le Charlier d'avoir accepté d'être notre promoteur de ce mémoire

Monsieur Abdelaziz Mounji pour son aide précieuse dans la maîtrise d'ASAX et ses conseils de lecture

Herr Professor Doktor Klaus Brunnstein et Frau Doktor Simone Fischer-Hübner pour leur chaleureux accueil dans le département informatique de l'université d'Hambourg ainsi que Morton Swimmer et Stefan Tode pour toutes les recherches et discussions menées ensemble à propos des virus

Nos parents respectifs pour nous avoir permis d'entamer ces études et nous avoir soutenus durant toutes ces années d'études

Madame Delhez pour avoir revu le côté grammatical de ce mémoire

Toutes les personnes qui nous ont directement ou indirectement aidés, conseillés ou critiqués dans la conception de ce travail.

Table des matières

1. Introduction

A tout moment, l'homme peut contracter un virus. Son système d'immunisation peut alors combattre rapidement avec succès certains de ces virus, mais d'autres peuvent conduire à la mort.

Il en est de même pour certains ordinateurs. A tout moment, ceux-ci peuvent être infectés par un virus. Certains peuvent être très amusants : on entend l'hymne national américain sur son ordinateur, une ambulance traverse l'écran, les caractères sur l'écran tombent comme des feuilles mortes. Mais si un nombre change dans la feuille de paie de vos employés, cela devient embarrassant. L'ordinateur est « malade ». Pour le soigner et le débarrasser des virus, on utilise des outils appropriés tels que les scanners ou d'autres programmes de détection et suppression de virus.

Ce qui a commencé par une blague d'étudiants devient de plus en plus criminel. Au mois d'avril de cette année, une enquête[1] menée en Allemagne a fait ressortir quelques chiffres assez effrayants. 81 % des entreprises en Allemagne ont subi au moins une infection par un virus en 2 ans et 13 % en ont subi plus de dix. Les coûts causés par une telle infection se situent une fois sur deux (58 %) entre 100 et 500 DM (i.e. entre 2.000 et 10.000 FB) mais plus de 13 % des infections coûtent plus de 2.000 DM (40.000 FB) aux entreprises.

De plus, dans 40 % des cas, le virus est seulement découvert sur le système après plus d'une semaine. Il a donc eu bien le temps d'infecter tout le système et même les sauvegardes (*backups*) du système.

Une des lois de Murphy sur les virus informatiques dit : « *If a virus can be written, it will be written* ».
Il y ajoute une deuxième loi : « *If a computer virus just cannot be written, it will be written anyway ; it will just take a little bit longer* ».
Ces deux lois se vérifient. Lors de notre stage, nous sommes partis au *Fachbereich Informatik, Universität Hamburg*. Nous avons travaillé au VTC, *Virus Test Center*. Notre maître de stage y était le professeur Klaus Brunnstein. Dans ce centre, ils connaissent plus de 8.500 virus pour PC et chaque semaine, ils en reçoivent plus de 50. Cela est principalement dû à la professionnalisation des auteurs de virus qui, face aux produits construits pour détruire leurs créations, recherchent constamment de nouvelles techniques pour que leurs virus soient encore plus discrets et moins vite trouvés.

De leur côté, les producteurs d'outils anti-virus ont de grandes difficultés à développer des outils satisfaisants. Le nombre de nouveaux virus qui naissent chaque mois est estimé à une petite centaine en moyenne. Il faut donc constamment mettre à jour ces outils. Une autre difficulté pour les outils anti-virus est de trouver des virus inconnus.

Les nouvelles stratégies d'infection utilisées par les virus, par contre, apparaissent plus rarement, en moyenne moins d'une par année. Il semble donc qu'il y ait tout intérêt à chercher également dans cette voie-là.

---

[1] Enquête trouvée dans le CD-ROM accompagnant la revue *PCgo!*, n° 5/96, Editions Markt & Technik

Devant le nombre important de virus reçus chaque semaine au centre, il serait bien de développer un outil permettant une classification plus rapide des virus pour faciliter le boulot aux chercheurs.
Au VTC, les chercheurs ont déjà développé un outil permettant de générer des fichiers audit-trail à partir d'une émulation PC, i.e. des fichiers reprenant tous les événements qui se sont produits dans une période déterminée sur un PC donné. Ces fichiers sont de taille immense et ne peuvent être analysés manuellement.
L'Institut d'Informatique de Namur, quant à lui, a développé un outil puissant appelé A.S.A.X., qui permet l'analyse dynamique de ce genre de fichiers importants à l'aide du langage RUSSEL. Lors d'une première collaboration entre les deux universités, un étudiant[2] avait été envoyé en stage pour permettre aux deux outils de communiquer.

Notre stage correspond à la deuxième collaboration. Nous y allions dans le but de créer de nouvelles règles génériques pour la détection de virus, i.e. de conceptualiser des stratégies d'infection utilisées par les virus sous forme de diagrammes pour les traduire ensuite en programme ASAX. Un étudiant hambourgeois, Morton Swimmer, avait déjà créé une première règle qui détectait déjà 95 % des virus les plus répandus sur le marché, montrant ainsi qu'il n'existe pas beaucoup de stratégies d'infection. Nous devions essayer d'en découvrir le plus avec de nouvelles règles.

Notre mémoire est organisé comme suit :
Dans le deuxième chapitre, nous allons donner une introduction au monde des virus informatiques et des outils existants pour les contrer.
Le chapitre 3 nous permettra d'approfondir les connaissances dans le système d'auditing PANDORA utilisé au stage.
Dans les chapitres 4 et 5, nous présenterons l'outil ASAX permettant l'analyse dynamique de fichiers audit-trail et l'extension par les tables associatives que nous avions développée à cet outil avant de partir en stage.
Dans les chapitre 6 et 7, nous entrerons dans le cœur du développement effectué à Hambourg et nous y présenterons les différentes possibilités de représentations par diagrammes des schémas d'attaque et les méthodologies de développement de ces diagrammes en programmes ASAX via le langage RUSSEL.
Dans le huitième chapitre, nous présenterons une erreur découverte lors de notre stage dans l'outil ASAX, la solution qui nous semble appropriée et un enrichissement possible à l'outil.
Dans le chapitre 9, nous montrerons les résultats obtenus grâce au programme développé en stage à l'aide de quelques exemples représentatifs.
Dans la bibliographie, vous trouverez les articles et livres que nous avons lus durant notre stage.
Dans les annexes se trouvent le code de notre programme d'analyse, la syntaxe des éléments du langage RUSSEL et la description des fichiers audit-trail générés par PANDORA.

[2] Voir **Vincent Haulotte,** *Etude des virus informatiques et utilisation d'un langage d'analyse d'audit-trail pour leur détection*, 1993

2. Virus informatiques

2.1 Définitions et concepts

Une bonne définition générale du virus reconnue par la communauté des chercheurs est celle donnée par Fred Cohen : « A computer virus is a program that can infect other programs by modifying them to include a possibly evolved copy of itself ». En français, un virus est un programme qui peut infecter d'autres programmes en les modifiant pour y inclure une copie - peut-être modifiée - de lui-même. Le terme « virus » utilisé pour désigner ces programmes provient des virus biologiques (voir section 2.2.).

Le terme « virus » est passé dans le sens usager, alors que parfois, il peut dénoter la présence d'un ver (*worm*) ou d'un cheval de Troie (*Trojan horse*).
Expliquons ces deux derniers termes.

Un ver (*worm*) est un programme qui envoie des copies de lui-même à d'autres systèmes informatiques via les connexions de réseaux. C'est donc un programme à part entière et se propage généralement par le courrier électronique ou des fonctions similaires. Contrairement à un virus, le ver ne requiert pas de programme hôte pour se propager.
Le ver le plus connu sur gros ordinateur est probablement l'Arbre de Noël qui a paralysé le réseau mondial d'IBM le 25 décembre 1987. Cet Arbre de Noël peut se propager dans les installations VM/CMS. Le programme invite l'utilisateur à taper « CHRISTMAS », dessine un arbre de Noël sur les terminaux des utilisateurs et s'envoie à tous les correspondants de l'utilisateur figurant dans les fichiers d'utilisateur NAMES et NETLOG.

Un cheval de Troie (*Trojan horse*) est un programme conçu pour effectuer certaines fonctions inconnues de l'utilisateur et non spécifiées dans la documentation. Dans ces fonctions inconnues peuvent se trouver des bombes logiques (code malicieux inséré dans un programme qui sera activé automatiquement lorsque certaines circonstances seront réalisées).
Prenons un premier exemple simple : le petit fichier batch suivant, appelé « SEX.BAT » illustre notre cas. **N'essayez surtout pas ce programme car il supprime la totalité des fichiers figurant dans le répertoire racine sur disque dur.** Il est cependant intéressant d'en connaître le fonctionnement :

```
DEL <SEX.BAT C:\*.*
Y
```

Cette séquence transfère l'entrée de la commande DEL de la console au fichier SEX.BAT qui contient également la réponse « oui » à la question du DOS : « Etes-vous sûr ? ».
Si l'utilisateur remarque ce fichier « SEX.BAT » au titre accrocheur sur une disquette et tape simplement « SEX » pour voir ce que fait la commande, tous les fichiers du répertoire racine de son unité C : (habituellement le disque dur) seront supprimés. On peut voir dans cet exemple que des programmeurs expérimentés et malveillants peuvent provoquer des dégâts nettement plus graves.

Citons un second exemple authentique : le 11 décembre 1989, 20.000 enveloppes contenant une disquette marquée « AIDS Information version 2.00 » accompagnée d'un guide étaient postées à Londres. Ce guide invitait le destinataire à mettre en place la disquette et à installer le logiciel. Au verso du guide figurait en petits caractère, l'indication « Licence Agreement » qui demandait à l'utilisateur d'envoyer 289 $ ou 378 $ pour pouvoir utiliser le logiciel. Cet accord

précisait qu'il y aurait des représailles si cette redevance n'était pas versée. Dès que l'utilisateur imprudent installait le logiciel, le programme imprimait une « facture » donnant l'adresse au Panama à laquelle le paiement devait être effectué. Ce programme AIDS, qui se fait passer pour un programme légitime fournissant des informations sur le SIDA, est censé déterminer le groupe de risque de l'utilisateur après lui avoir demandé de remplir un questionnaire. Or, la procédure d'installation modifie le fichier AUTOEXEC.BAT qui entraîne à chaque exécution de ce fichier l'incrémentation d'un compteur situé dans un fichier caché. Lorsque ceci se produit un certain nombre de fois (environ 90), la séquence de détérioration se déclenche. L'utilisateur est invité à attendre, pendant que la majorité des noms des fichiers sur disque dur sont cryptés (brouillés) et signalés « cachés ». Le seul fichier non caché contient un message leur expliquant qu'il serait sérieusement temps de penser à payer pour la licence. Il s'agit là d'un exemple type de tentative d'extorsion s'appuyant sur l'utilisation d'un cheval de Troie. On demande tout d'abord à l'utilisateur d'installer le logiciel (qu'il est difficile de désinstaller) puis on lui fait du chantage pour qu'il paie afin de ne pas avoir d'ennuis.

Les virus informatiques peuvent être définis par quatre critères essentiels :

1. la reproduction : les virus se reproduisent, investissent les disquettes, les systèmes informatiques et les réseaux.
2. le chemin exécutable : un virus ne peut faire quelque chose que s'il est exécuté. Cette exécution se fait normalement de façon parasite et l'utilisateur ignore donc que le virus a été exécuté. Le système d'exploitation, qui s'exécute automatiquement au lancement et les applications couramment exécutées, sont tous les deux, du point de vue du virus, de bons « vecteurs » pouvant transporter gratuitement le virus.
3. les effets secondaires : en général, les virus ne sont pas uniquement constitués d'une codification autoreproductrice ; ils contiennent également une charge. On peut facilement programmer les effets secondaires de la charge pour qu'ils soient malveillants.
4. le camouflage : la bonne propagation d'un virus dépend du délai pendant lequel il peut se reproduire impunément, avant que sa présence ne soit trahie par l'apparition d'effets secondaires. La durée de la reproduction s'obtient par deux modes de camouflage : le cryptage (brouillage) qui permet au virus de faire paraître sa codification différente pour chaque application contaminée (rendant ainsi difficile la recherche d'une configuration fixe pour le virus) ; et l'interception des interruptions. Les applications PC-DOS utilisent des interruptions logicielles pour communiquer avec le système d'exploitation. Lorsqu'une application émet une interruption, un branchement se produit à une adresse prédéterminée. Si un virus change une ou plusieurs de ces adresses, tout branchement sur le système d'exploitation peut passer par ce virus, qui peut alors décider de ce qu'il veut faire d'une demande particulière.

On peut ajouter qu'après une infection par un virus, la plupart du temps, le programme attaqué devient un cheval de Troie. En effet, même si, au départ, ce programme n'était pas un cheval de Troie, après l'infection par le virus, le programme sera bien devenu un cheval de Troie puisqu'il contiendra de nouvelles fonctionnalités inconnues de l'utilisateur.
Quand ce programme infecté sera exécuté, le virus est également exécuté, propageant ainsi l'infection, normalement de manière invisible pour l'utilisateur.

Un virus ne peut infecter un ordinateur sans assistance, il est propagé par des vecteurs souvent humains. Dans le système, le virus ne peut rien faire d'autre que se propager lors d'exécutions de programmes déjà infectés. Cependant, généralement, après sa propagation, ce virus commence à faire des actions, comme écrire certains « gentils » messages sur le terminal ou

jouer des blagues étranges sur l'écran. Des virus plus méchants peuvent également faire des dommages irréversibles, comme effacer tous les fichiers de l'utilisateur.

C'est vers les années 1990 que les virus sont devenus un sérieux problème spécialement parmi les utilisateurs de PC IBM et de Macintosh à cause du manque de sécurité sur ces machines permettant aux virus de se propager facilement, en infectant même le système d'exploitation.

La production de programmes anti-virus spéciaux est devenue une industrie et un nombre important de reportages exagérés effectués par les médias ont provoqué chez les utilisateurs une certaine hystérie. Beaucoup de gens aujourd'hui ont tendance à blâmer tout ce qui ne marche pas comme étant dû à une attaque de virus.

2.2 Analogie avec les virus biologiques

Un virus informatique est un petit segment de code qui est incapable de se reproduire sans être incorporé dans un programme hôte. Ce code peut être comparé à la structure DNA du virus biologique. L'analogie avec le programme hôte est évidente : dans le cas du virus biologique, il y a injection d'un matériel génétique dans une cellule, et dans le cas du virus informatique, injection de code dans un programme d'ordinateur.
Il y a également d'autres analogies possibles :

- les portes d'entrée : le nombre varié d'entrées possibles du virus biologique dans le corps humain peut être comparé au grand nombre d'entrées par lesquelles le code du virus informatique peut être introduit dans un système.
- les vecteurs : les vecteurs sont des organismes qui portent l'infection virale (la mouche tsé-tsé). Pour une infection informatique, les vecteurs sont des données ou des programmes.
- l'hygiène : pour se prévenir d'une infection, une certaine quantité de mesures d'hygiène sont recommandées. Le parallèle serait d'éviter l'intégration de code suspect (des softwares anonymes) et de vérifier régulièrement l'intégrité des softwares.
- la vaccination : c'est une technique puissante d'immunisation naturelle utilisant un virus atténué. Dans les ordinateurs, on peut également vacciner l'ordinateur pour empêcher l'entrée d'un virus en utilisant des strings de reconnaissance de virus (voir les outils d'inoculation dans la section 2.5.4.).
- les anticorps : les anticorps à des infections spécifiques sont équivalents à l'introduction dans l'environnement du système de logiciels de désinfection spécifiques. De plus, quand l'infection par le virus biologique est détruite, le nombre d'anticorps dans le corps diminue, de même, la probabilité que les administrateurs du système restent consciencieux dans l'emploi de programmes de scanning et de désinfection diminue.
- l'isolation et la quarantaine : dans une communauté, lors de maladies hautement infectieuses, on suggère d'isoler les organismes infectés. De même, pour réduire le risque d'extension de l'infection d'un virus informatique, il est conseillé de couper le trafic entre les systèmes infectés et propres.
- l'incubation : beaucoup de maladies ont un délai entre le moment de l'infection et le moment où l'organisme montre les symptômes de l'infection. On peut comparer cela aux virus informatiques avec le délai entre l'infection du système par le virus et le commencement de l'activité malicieuse de ce dernier.
- le diagnostic : le processus de diagnostic d'un système infecté peut être comparé à celui d'un organisme infecté. Des signes externes contribuent au diagnostic ou des vérifications internes peuvent être exécutées (similaire à l'inspection des données sur un disque ou dans la mémoire).

2.3 Types de virus

On peut classer les virus de manière générale en deux catégories: les virus-système (*system virus*) et les virus-fichiers (*file virus*). A ces deux catégories s'ajoutent encore des virus hybrides : les virus multi-partite et virus système-de-fichiers (*file system virus*). Il est à préciser que, la plupart du temps, ces types ne sont à envisager que sur les systèmes DOS. Bien sûr, certains de ces types existent sur d'autres systèmes ; par exemple, les virus compagnons existent également dans les systèmes UNIX.

2.3.1 Virus système

Les virus système infectent les parties système d'un support de stockage en utilisant l'interface machine (BIOS). Le système d'exploitation n'est pas encore chargé en mémoire à ce moment-là.
Presque tous les virus système connus s'attaquent au secteur de démarrage (*boot sector*) et/ou secteur de partitions (MBR - *Master Boot Record*). Le MBR est le premier secteur sur un disque dur et doit exister pour qu'on puisse accéder au disque dur. Dans ce secteur se trouve la table des partitions. Toutes les partitions du disque dur y sont énumérées. Les partitions sont des parties d'un disque dur qui peuvent être accédées comme des drives virtuels. Chaque partition peut contenir un système d'exploitation différent. Une partition doit être déclarée «active» pour qu'on puisse démarrer le système à partir d'elle. Toutes les partitions contiennent un secteur de démarrage qui est toujours en premier lieu. Le secteur de partition contient du code exécutable, qui charge le secteur de démarrage de la partition active et l'exécute.

2.3.1.1 Virus MBR

Le virus MBR infecte un secteur de partition en le déplaçant et en le remplaçant. Ce faisant, le virus est exécuté le premier quand on démarre le système du disque dur. Après s'être chargé en mémoire, il appelle le secteur original et continue la séquence normale de démarrage.

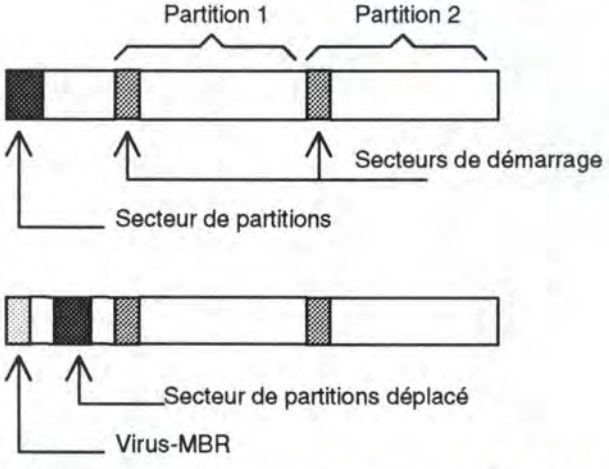


Fig. 2.1 : Schéma d'un disque dur, avant et après infection par un virus MBR

2.3.1.2 Virus «secteur de démarrage»

Un secteur de démarrage sert à démarrer le système d'exploitation stocké sur le support informatique. Ce secteur contient du code pour initier le chargement du système. Tous les supports possèdent ce secteur, même les disquettes qui servent uniquement à contenir des données.

Puisque ce secteur est chargé automatiquement en mémoire quand on lance le système, il est une source importante d'infection. En fait, dans des conditions normales, un démarrage avec un secteur de démarrage vide provoque simplement une erreur du style « *Veuillez introduire un support valide de démarrage* ». Un virus peut alors profiter de cette situation pour exécuter sa petite routine et faire comme si rien ne s'était passé.

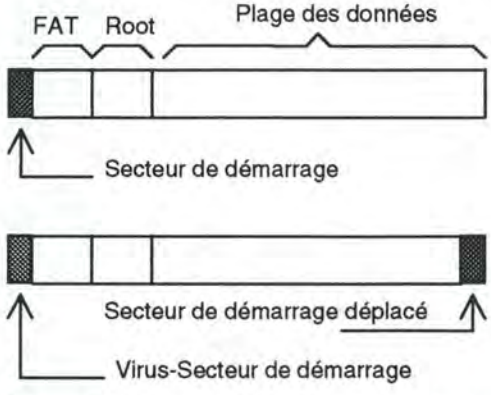


Fig. 2.2 : Schéma d'une disquette (ou partition), avant et après infection par un virus système (Bootsector)

Le secteur original est déplacé et remplacé par une partie du virus. Le code restant du virus est enregistré dans la partie données du support. Pour le protéger contre effacement, certains virus déclarent alors ces emplacements comme étant endommagés («BAD»). Quand l'ordinateur est alors démarré à partir de cette disquette ou partition, le virus est chargé d'abord. Il se charge complètement en mémoire et y reste résident, puis donne la main au secteur original.

2.3.2 Virus fichiers

Les virus fichiers infectent des fichiers exécutables. Sous DOS, les exécutables portent le plus souvent les extensions COM ou EXE, mais d'autres sont également possibles (par exemple les fichiers BAT - exécution en batch, ou les DLL (*Dynamic Link Library*) sous Windows, qui comportent un corps exécutable). Il se peut que des fichiers contenant seulement des données soient touchés, mais puisqu'ils ne sont jamais exécutés ou interprétés, une «infection» aurait alors comme seul effet une destruction des données.

2.3.2.1 Fichiers COM

Les fichiers COM sont des fichiers directement exécutables, c'est-à-dire qu'ils sont chargés entièrement en mémoire et exécutés à partir du premier byte du code. Pour des raisons historiques (ils sont construits de la même manière que les vieux programmes CP/M), leur taille est limitée à 64 KByte. Pour infecter un fichier COM, le virus s'enregistre par exemple à la fin du fichier et change la première instruction vers un saut sur le corps du virus. Ainsi il est assuré que le virus est exécuté en premier lieu quand on exécute un programme.
Dans la Fig. 2.3., on peut voir qu'après exécution du code, celui-ci fait un saut vers le début du programme hôte pour cacher sa présence.

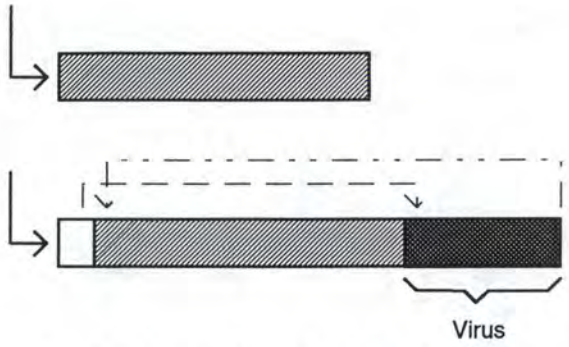
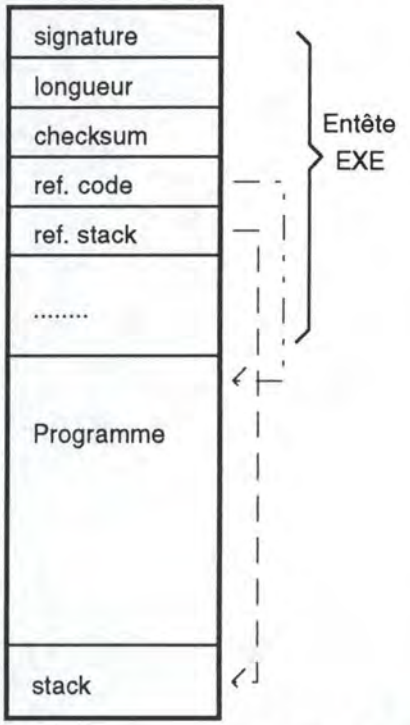


Fig. 2.3 : Infection d'un fichier COM

Des virus qui attaquent les fichiers COM doivent respecter la limite des 64 KBytes; ils sont donc parfois amenés à utiliser des techniques de compactage pour rester dans la plage autorisée. De plus, sur les trois premiers bytes, se trouve toujours un pointeur vers le début du code du programme, c'est ce pointeur que le virus change la plupart du temps.

2.3.2.2 Fichiers EXE

L'infection des fichiers EXE se fait d'une manière analogue à celle des fichiers COM, mais la construction d'un fichier EXE est différente.



- La signature d'un fichier EXE est une zone de 2 bytes qui contient soit «MZ» ou «ZM»

- La longueur du fichier est exprimée en nombre de pages de 512 bytes utilisées et nombre de bytes sur la dernière page partiellement remplie.

- Le checksum n'est pas obligatoire, car DOS ne le vérifie pas.

- Il y a une référence vers le début du code du programme,

- également l'adresse du stack dans le programme.

- ... et encore d'autres renseignements sur la table de relocation et sur une entête étendue (pour les programmes Windows).

Fig. 2.4 Construction typique d'un fichier EXE

Lors d'une infection, l'adresse du début de code est modifiée dans le sens où elle pointe maintenant sur le code du virus. La valeur de la taille est également adaptée.

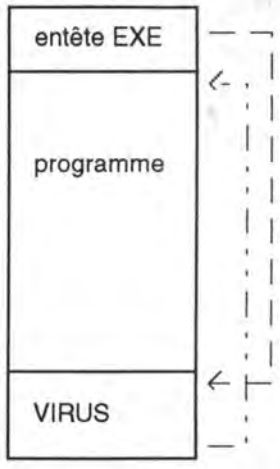


Fig. 2.5 : Infection d'un fichier EXE

Comme pour les fichiers COM, la plupart du temps, les virus changent cette en-tête et transforment alors cette zone de 26 bytes pour y mettre les nouvelles caractéristiques du fichier.

2.3.3 Virus système-de-fichiers

Un virus système-de-fichiers n'infecte pas directement les fichiers, mais attaque le système des fichiers du système d'exploitation. Il travaille au niveau machine et modifie les structures du système.
A cause de la complexité de cette technologie, on trouve relativement peu de virus dans cette catégorie. En plus, ces virus ne marchent que dans des systèmes d'exploitation qui autorisent un accès direct en lecture ou en écriture au disque.
Comme exemple, on peut citer un représentant typique de cette classe: le virus DIR-II. Il se place dans un cluster sur disque et modifie toutes les entrées des répertoires pour fichiers exécutables pour que l'entrée du premier cluster pointe sur le cluster du virus. Il obtient ainsi la garantie qu'il est exécuté d'abord. Quand il est actif en mémoire, il redirige le pointeur sur le premier cluster original. Le fichier original n'a jamais été touché, mais son exécution provoque le lancement du virus.

2.3.4 Virus multi-partite

Quand un virus possède les caractéristiques de deux ou plusieurs types de virus, alors il est appelé virus multi-partite.
Un exemple en est le virus Tequila, qui infecte et le secteur de partitions et des fichiers. Quand un fichier infecté est exécuté, le virus essaie d'infecter seulement le secteur de partitions. Après le prochain redémarrage, il s'attaque aux fichiers pour garantir d'être résident en mémoire avant tous les autres programmes.

2.3.5 Virus compagnons

Un virus compagnon utilise une caractéristique des systèmes d'exploitation : l'existence d'un certain ordre dans lequel les fichiers sont cherchés sur le disque par le système.
Dans le cas du DOS, quand un utilisateur tape un nom de fichier à exécuter, l'interpréteur des commandes regarde d'abord dans le répertoire courant après un fichier ayant l'extension COM, puis EXE et BAT s'il n'a pas trouvé de correspondance. S'il ne trouve toujours rien, il regarde dans tous les répertoires spécifiés dans le «path», suivant le même ordre de recherche des extensions.

Dès lors, pour infecter un fichier, le virus n'a qu'à créer un fichier ayant une extension avec une position antérieure dans l'ordre des extensions. Le virus ainsi peut se propager et n'a pas modifié le programme qu'il voulait infecter.

2.3.5.1 Compagnons «réguliers»

Cette procédure de recherche peut être utilisée par un virus. Il peut localiser un fichier avec l'extension EXE et mettre le corps du virus dans le même répertoire avec le même nom, mais avec l'extension COM. Comme ça, le virus est exécuté à la place du programme voulu. Il est alors en mesure «d'infecter» d'autres fichiers de la même manière. Après, il rend la main au programme EXE dont il vient de créer le fichier compagnon.
Ces compagnons sont relativement faciles à détecter, mais rien n'empêche le virus de se cacher, c'est à dire mettre l'attribut «caché» pour le fichier compagnon. S'il est en plus résident en mémoire, il pourrait intercepter des appels système et cacher la présence du fichier.

2.3.5.2 Compagnons «chemin»

Au lieu de mettre le corps du virus dans le même répertoire que le fichier original, il peut être placé avec n'importe quelle extension d'un exécutable dans un répertoire qui se trouve antérieur dans la liste des répertoires du «path». Ceci aura le même effet que les compagnons réguliers, mais c'est moins facile à voir. Pour détecter ce genre d'attaque, il faudrait passer en revue tous les répertoires (ou au moins ceux du «path») pour découvrir des doubles, i.e. des fichiers de même nom mais avec une extension différente.

2.3.5.3 Compagnons «alias»

Une autre technique consiste en la définition d'alias. Les alias sont des petites macros. Les commandes dans ces alias sont exécutées avant tout fichier exécutable.
Par exemple, on pourrait définir un alias qui s'appelle DIR et qui exécute d'abord un programme quelconque, avant de faire effectivement ce qui a été demandé, c'est-à-dire afficher le contenu du répertoire.
Ce n'est de nouveau pas trop dur à détecter, car ces alias doivent être définis quelque part dans un fichier de configuration.

2.3.6 Virus Macro

Les virus macro sont la dernière invention des auteurs de virus. Avec le développement de langages macro toujours plus puissants, par exemple WordBasic dans MS-Word et Visual Basic for Applications dans MS-Excel, on leur fournit un moyen de haut niveau pour infecter des systèmes. Les langages Macro permettent la rédaction de vrais programmes, incluant des fonctions d'appel d'autres programmes. Les macros peuvent être déclarées de telle sorte qu'elles sont automatiquement exécutées quand on ouvre un texte Word ou une feuille Excel. De plus, ils sont indépendants du système d'exploitation, car ils utilisent une application standard pour se lancer. Par exemple, Word existe pour DOS et Macintosh ; dès lors, le virus macro peut tourner sur différentes plates-formes.
Une faiblesse de ces virus est actuellement la nécessité de devoir retrouver le même type d'environnement, c'est-à-dire qu'une macro écrite en anglais ne peut pas s'exécuter dans un Word français.

2.4 Attributs des virus

2.4.1 Virus réinscripteur *(overwriting virus)* - non-réinscripteur

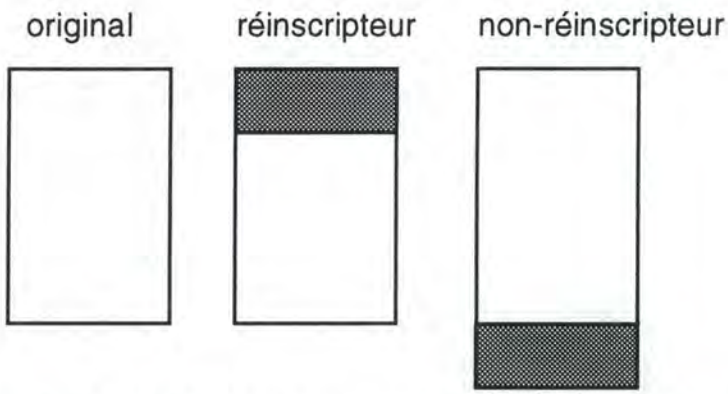


Fig. 2.6 : Situation du fichier hôte avec virus réinscripteur et non-réinscripteur

Quand un programme est endommagé lors de l'infection et que seul le virus est capable de s'exécuter, alors le virus s'appelle réinscripteur. Le virus essaie de ne pas se faire remarquer par un changement de taille du fichier hôte.
Lorsque le programme n'est pas endommagé, c'est-à-dire que l'hôte s'exécute comme avant, le virus s'appelle non-réinscripteur.

2.4.2 Infection directe - indirecte

Des virus infectent directement si lors du lancement d'un fichier infecté, le virus essaie de gagner le contrôle du système en cherchant activement un objet infectable. Ces virus ne restent pas en mémoire et perdent le contrôle quand ils le passent au programme hôte.
Ceci est une technologie assez facile à utiliser et bien maîtrisée par les programmeurs de virus. Par contre, ces virus ne sont pas très répandus, puisqu'ils ne se répandent que très lentement, surtout sur disquettes. Une exécution trop lente augmenterait le risque de détection. C'est pour cela que ces virus infectent toujours de manière sélective:

- Les victimes sont cherchées dans le répertoire courant. Presque tous les virus directs travaillent de cette façon.
- Un virus pourrait visiter tous les répertoires spécifiés dans le «path». Certains virus de la famille «Vienna» agissent ainsi.
- Très peu de virus passent en revue tous les répertoires à la recherche de leurs victimes.

En ce qui concerne l'infection indirecte, le virus infecte d'abord le système résident, i.e. il réserve de la mémoire pour lui et change le fonctionnement du système (détournement de certaines interruptions ...). Alors, à l'occurrence de certains événements définis par le virus, des objets (fichiers, secteur de démarrage, MBR, ...) seront infectés. En principe, toutes les fonctions système pourraient être utilisées. Sont très utiles les fonctions sur les fichiers, car elles travaillent directement avec un fichier existant. Des fichiers peuvent alors être infectés lors de l'utilisation d'une fonction de lecture, copie...
Par exemple, un virus pourrait capter l'appel de la fonction «exécution de fichier» et infecter tous les fichiers qui sont exécutés par cette fonction.

Il y a des virus qui ne s'activent que lors d'une écriture. Ils sont appelés *Slow Infectors*. Ils sont ainsi immunisés contre la méthode anti-virus qui utilise les checksums. Ces checksums sont recalculés automatiquement lors d'une modification «voulue» (voir section 2.5.1.3.)

2.4.3 Les virus «Stealth»

Un virus possède une propriété *stealth*, quand il utilise des mécanismes de camouflage pour ne pas être détecté. Quand il est actif en mémoire, il intercepte des appels aux objets infectés et fait en sorte que les résultats fournis fassent apparaître les objets comme non infectés.
Il y a certaines classes de caractéristiques stealth, que l'on retrouve le plus souvent pour des virus-fichiers.
Ce type de virus n'est pas trop difficile à développer quand on sait utiliser les faiblesses des PC. Le PC n'a pas été développé et conçu de la même manière que les mainframes par exemple. Ces derniers ont été conçus dans un esprit de sécurité pour que les données traitées par ces grosses machines ne puissent être altérées. Par contre, le PC a été développé pour l'utilisateur normal et on ne pensait pas au début, que l'ordinateur individuel allait se trouver aussi vite dans presque tous les foyers et entreprises. Le PC a beaucoup de faiblesses : il n'y a pas de protection mémoire (il est facile pour un virus d'intercepter les interruptions logicielles pour se camoufler) ; un programme est une suite de bytes, ordonnés de manière rigoureuse, il suffit donc de connaître cette structure et on peut ainsi la transformer facilement sans que le système s'en rende compte.

2.4.3.1 Stealth - Attributs

Un virus cache sa présence en ne modifiant pas les attributs des fichiers, ou bien leur changement reste invisible.
Puisque sous DOS, il n'y a pas de mécanismes de contrôle d'accès aux attributs de fichiers, le virus peut sauvegarder les attributs avant infection et les restaurer après.

2.4.3.2 Stealth - Longueur

Le changement de taille des fichiers infectés est caché ou bien évité.
Il existe certaines méthodes par lesquelles un virus peut cacher le code qu'il a ajouté au fichier. La méthode la plus facile est de rechercher une zone de bytes constants dans le programme hôte et de s'implanter dans cette zone. Quand le programme est chargé en mémoire, le virus s'installe, s'exécute et rétablit l'ancienne valeur dans la zone qu'il avait occupée.
Une autre manière est de placer un genre de filtre devant les fonctions DOS. Quand le virus reçoit une requête lui demandant de renvoyer la taille du fichier, quand il a affaire à un fichier infecté, il soustrait la taille du virus de la vraie taille du fichier.

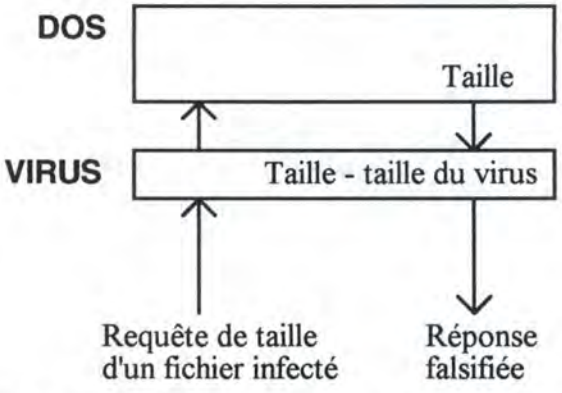


Fig. 2.7 : Fonctionnement d'un virus «stealth»

Les deux méthodes ont pour conséquence, que l'utilisateur ne peut pas se renseigner sur la vraie taille avec des moyens standards (DIR, ...).

2.4.3.3 Stealth - Contenu

Lors d'une lecture d'un fichier infecté, le virus est enlevé. Comme pour le virus stealth-longueur, un filtre est installé, qui ne cache pas seulement les différences de taille, mais également le contenu du fichier infecté. Le fichier est alors désinfecté en mémoire, et le lecteur ne remarquera aucun changement vis-à-vis de l'original.

2.4.4 Chiffrage

Si un virus garde certaines parties de son code chiffré, alors il s'appelle un virus chiffré.
Ceci sert principalement à rendre plus difficile toute contre-mesure et/ou pour cacher des messages suspects envers ceux qui pourraient regarder le code (certains virus affichent un message à l'écran; on veut alors empêcher de pouvoir le retrouver tel quel dans le code source du programme).
Un virus est complètement chiffré si toutes ses parties, sauf le décrypteur, sont chiffrées. Il est évident que le décrypteur ne peut être chiffré lui aussi, car il est exécuté en premier et se charge de déchiffrer les autres parties.
Si seulement certaines parties sont chiffrées, il est dit *partiellement chiffré*. Ceci ne présente pas un grand intérêt car cette partie du code pourrait être utilisée comme signature du virus par n'importe quel scanner (voir section 2.5.).

2.4.5 Polymorphisme

Quand un virus polymorphique se reproduit, la copie est équivalente du point de vue fonctionnalité, mais le code n'est plus le même. Pour arriver à ce résultat, le virus peut insérer aléatoirement des instructions superflues dans le code (par exemple `SKIP`, assignation `a:=a`, ...), changer l'ordre des instructions indépendantes, utiliser des instructions différentes qui produisent le même effet net (par exemple Soustraire `a` de `a` ou `a:=0`, ...), ou bien utiliser un des nombreux schémas de chiffrement.
Ce mécanisme à été inventé pour entraver la reconnaissance des signatures de virus, i.e. des parties caractéristiques dans le code d'un virus.

2.4.6 Tunneling

On appelle « tunneling » la tentative par les virus de ne pas se faire remarquer par des programmes qui supervisent les appels système. Ces virus s'installent de telle manière qu'ils contournent les filtres de détection en parlant directement avec le système d'exploitation. C'est ainsi qu'ils espèrent rester inaperçus.

2.5 Outils de lutte anti-virus

Les outils anti-virus réalisent trois fonctions de base : ils détectent, identifient ou suppriment les virus. Quelques outils empêchent l'infection par un ou plusieurs virus, on parle alors d'inoculation, mais l'application de ces outils est limitée.

2.5.1 Outils de détection

Ces outils détectent l'existence d'un virus sur un système. Les virus peuvent être détectés avant exécution, durant l'exécution ou après exécution et reproduction. Les virus peuvent être détectés dans différents points du système car ils peuvent être en cours d'exécution, résidents en mémoire ou stockés en mémoire.

2.5.1.1 Détection par analyse statique

Ces outils examinent les exécutables sans les exécuter. Ils peuvent être utilisés de deux manières :

1. ils détectent du code infecté avant son introduction dans le système en testant toutes les disquettes avant d'installer le logiciel sur le système ;
2. ils testent le système de manière fréquente pour détecter tout virus acquis entre les phases de détection.

Les détecteurs par analyse statique utilisent surtout les techniques de recherche de signature (*signature scanning*) et de détection algorithmique (*algorithmic detection*). Ces outils sont plus connus sous le nom courant de **scanners**.

Les scanners peuvent détecter le code d'un virus dans un exécutable ou même détecter un virus résident en scannant la mémoire. Ils sont limités intrinsèquement à la détection de virus connus, mais peuvent parfois détecter de nouvelles variantes de certains virus.

Dans la technique de recherche d'un virus par signatures, des séquences de code binaire sont recherchées dans les exécutables. Chacune de ces séquences est propre à un virus ou à une famille de virus, séquences trouvées par les concepteurs de l'outil en étudiant attentivement chaque virus. Cette signature est la signature que le virus recherche au moment de l'infection pour savoir s'il a déjà infecté un fichier ou non. Toutes ces signatures sont regroupées dans une base de données accessible à l'outil. Dans cette technique, on utilise également le concept de la position relative de la signature, ce qui accroît la fiabilité du scanner.

Dans l'approche théorique développée par Fred Cohen, il démontre qu'il sera toujours impossible d'avoir un programme qui détecte 100 % des virus. Les scanners ont comme gros problème qu'il faut remettre constamment la base de données à jour. Certains auteurs estiment que le plus récent des anti-virus ne reconnaîtra jamais plus de 95 % des virus actuels, car de nouveaux virus apparaissent chaque jour. Par exemple, dans sa dernière version anti-virus, F-PROT prétend détecter 98 % des virus existants, mais depuis, combien de virus ont été développés ?

Les virus polymorphiques n'ont pas de signatures fixes puisqu'ils s'auto-modifient et sont souvent chiffrés aléatoirement. Si on veut les repérer, il faut donc les comparer à une multitude de signatures, c'est pourquoi une recherche avec une méthode algorithmique est plus complète et plus puissante. Cette méthode consiste en la recherche de l'algorithme type du virus, l'algorithme d'infection général, même si le code en lui-même est constamment changé et souvent chiffré.

Enfin, les derniers outils développés dans ce domaine sont les détecteurs par analyse heuristique du code binaire. Ils recherchent dans le code de l'exécutable des techniques couramment utilisées par les virus et non utilisées par les programmes courants. Il peut détecter par exemple la présence dans un exécutable de code auto-chiffré ou bien un ajout de code à un programme existant.

2.5.1.2 Détection par interception

Pour se propager, un virus doit infecter d'autres programmes hôtes. Les outils de détection par interception stoppent des tentatives de réalisation de telles activités illicites, i.e. que ces outils arrêtent l'exécution de programmes infectés par le virus lorsque le virus essaie de se reproduire

ou de devenir résident. Il est à noter que le virus a déjà été introduit dans le système et essaie de se reproduire avant que la détection puisse avoir lieu.

On dénombre trois types d'outils de détection par interception : les contrôleurs de flux, les moniteurs (*general purpose monitors*) et les « shells » de contrôle d'accès (*access control shells*).

a) Les **contrôleurs de flux** sont des outils qui surveillent le flux de données entre les périphériques et la mémoire centrale. En fait, ce sont des extensions des outils de détection par analyse statique, toute la différence est dans le fait qu'ils résident tout le temps en mémoire et qu'ils recherchent dans les flux de données des signatures éventuelles de virus. Un gros avantage par rapport aux scanners est qu'ils peuvent détecter un virus avant l'infection de la machine et peuvent ainsi bloquer le transfert pour empêcher l'infection. Malheureusement, ils ne détectent pas les virus qu'ils ne connaissent pas et réduisent la vitesse des transferts de données et de plus, il y a toujours ce problème de mise à jour.

b) Les **moniteurs** sont actifs en permanence en mémoire et permettent la détection en temps réel des virus. Ils détectent les comportements qui leur semblent suspects ou ressemblant à un virus, provoquant alors un message d'alarme. Les concepteurs écrivent un modèle intégrant les comportements suspects, la façon de les détecter ainsi que les modules pour les contrer, modules qui sont ajoutés au système d'exploitation. Ne seront évidemment pas détectés les virus qui utilisent de nouvelles méthodes qui tombent hors du modèle. De plus, sur un PC, le système de protection de la mémoire n'est pas présent ou pas utilisé par le système d'exploitation ; ainsi, le système d'exploitation et le moniteur sont vulnérables car ils peuvent être contournés.

c) Les **shells de contrôle d'accès** sont intégrés au système d'exploitation. Contrairement aux moniteurs, ils ne détectent pas des comportements de virus mais tentent de renforcer le contrôle d'accès aux parties du système. Ils contrôlent les accès aux fichiers de données, aux exécutables et aux secteurs stratégiques des disques. Les accès illégaux ou les modifications de ces objets seront signalés par des messages d'alarme. Ces tentatives pourront même être neutralisées par le logiciel. Les shells peuvent également utiliser des systèmes de chiffrement rendant inaccessibles les informations vitales sans l'intermédiaire du shell.

2.5.1.3 Détection de modification

Tous les virus provoquent des modifications d'exécutables lors du processus de reproduction. Grâce à cela, la présence de virus peut être détectée par la recherche d'une modification inattendue d'exécutables. Ce processus est appelé vérification d'intégrité (*integrity checking*).

Il est à noter que dans ce cas, pour que l'outil de détection marche, il faut que les exécutables infectés soient introduits dans le système **et** que le virus se soit reproduit.

En théorie, les exécutables sont des objets statiques, dès lors, une modification est la traduction d'une infection par un virus. Ce n'est pas toujours vrai, il existe des programmes qui s'auto-modifient ou qu'il est nécessaire de recompiler à chaque exécution ; dans ces deux cas, la détection d'une modification n'est pas la solution pour la détection de virus.

La technique utilisée pour détecter les modifications d'exécutables est la technique du « checksum », à traduire par somme d'intégrité. Cette technique consiste à donner comme paramètre à une fonction mathématique un exécutable et à recevoir un résultat en sortie que

l'on appelle checksum. Cette fonction est appliquée une fois lorsque le système est sain (ou est supposé l'être). Ensuite, cette valeur est périodiquement recalculée. Une modification de la valeur indique une modification de l'exécutable.

La technique de détection de modification utilise deux techniques mathématiques : le contrôle de redondance cyclique (*cyclic redundancy checks*) et les checksums chiffrés (*cryptographic checksums*).

1. Le **contrôle de redondance cyclique** appelé communément CRC est une technique utilisée pour vérifier l'intégrité des données voyageant entre périphériques et mémoire centrale, ou dans un réseau, et pour tout type de communication entre ordinateurs. Malheureusement l'algorithme n'est pas très sûr car non conçu au départ contre les virus mais pour les transmissions. Ce CRC peut donc être cassé. De plus, le CRC ne saurait que détecter la reproduction du virus mais ne saurait l'identifier.
2. Le **checksum chiffré** est obtenu en appliquant un algorithme de chiffrement sur les bytes d'un exécutable. On peut utiliser des algorithmes à clé publique ou privée. La clé privée sert surtout pour l'efficacité, puisque l'application d'une clé privée est plus aisée.

Pour ces outils de détection, il faut s'assurer que l'outil n'est pas en mémoire en même temps qu'un virus *stealth* car il serait capable de donner une image saine du système et des exécutables déjà contaminés. Il faut donc démarrer les tests avec une machine à système d'exploitation sain.

2.5.1.4 Détection par analyse d'exécution

Les détecteurs par analyse d'exécution essaient de voir ce qui se passe de l'intérieur. Dans le cas du traçage d'exécution, l'outil essaie d'observer pas à pas l'exécution d'un programme après que cette exécution ait eu lieu tandis que les moniteurs faisaient l'analyse de l'exécution du programme au moment même de son exécution.

Pour les moniteurs, l'analyse se fait on-line tandis que pour les outils d'analyse par traçage d'exécution, elle se fait off-line. Pour conserver ces traces d'exécution, il faut évidemment d'autres outils. Dans la section 3.4.2., nous développerons les différentes technologies qu'il est possible de mettre en œuvre pour réaliser le traçage d'exécution pour le système DOS.

2.5.2 Outils d'identification

Les outils d'identification sont utilisés pour identifier quel virus a infecté un exécutable particulier. Cela permet à l'utilisateur d'obtenir une information supplémentaire sur le virus. On obtient ainsi des renseignements sur les autres types de dommages subis et les procédures de nettoyage appropriées du virus.

Pour cela, il existe des outils plus performants que les scanners fournis dans le marché qui sont les outils d'identification précise. En plus de la signature, il y a connaissance de valeurs de checksum réalisés sur les parties constantes d'un virus. Toutes ces données sont rangées dans une base de données. On comprend dès lors que la qualité d'un produit de détection dépend de la précision de sa base de données. C'est d'ailleurs un argument de vente de ces produits.

Lorsqu'on a détecté un virus et qu'on n'arrive pas à lui donner un nom, on a affaire à un nouveau virus. Transmis aux chercheurs et concepteurs d'anti-virus, ce virus pourra alors être suivi. Cela permettra de séparer les nouveaux virus des nouvelles variantes d'anciens virus par exemple.

2.5.3 Outils de suppression

Dans beaucoup de cas, une fois qu'un virus a été détecté, il se trouve déjà sur de nombreux systèmes ou dans de nombreux exécutables sur un même système. La récupération par les disquettes originales ou des backups propres (i.e. sans virus) peut être un processus pénible. Les outils de suppression essaient de restaurer efficacement le système dans son état non-infecté en extrayant le code du virus de l'exécutable infecté.

La méthode la plus sérieuse et la plus sûre pour la suppression d'un virus reste la suppression pure et simple des exécutables infectés et leur restauration à partir des copies de sauvegarde non infectées. Mais ce n'est pas toujours possible, un exemple typique est l'infection de secteurs stratégiques sur les disques durs. La plupart du temps, pour réinstaller ce genre de secteurs, il faut reformater les disques infectés et y réinstaller ensuite tous les logiciels. De plus, la restauration du système à partir des *backups* peut être encore plus risquée si le virus qui a infecté le système est un *slow infector*. En effet, s'il n'est détecté que plus tard, cela signifie qu'il y a de grandes chances que les sauvegardes du système effectuées précédemment soient également infectées.

Lorsque le code d'un exécutable a été complètement remplacé, l'infection est pratiquement incurable car la guérison ne garantit pas que l'exécutable ne comporte pas de séquelle et que l'exécutable guéri a la forme qu'il avait avant l'infection.
Pour effectuer une guérison, l'outil de suppression possède un module de guérison associé à chaque virus ; c'est pourquoi il est très important de bien identifier le ou les virus à l'origine de l'infection.

2.5.4 Outils d'inoculation

Dans certains cas, un exécutable peut être protégé contre un petit nombre de virus par inoculation. Cette technique consiste à insérer dans les exécutables à protéger la signature qu'utilise un virus pour se reconnaître. Cette signature doit être insérée à l'endroit approprié.

Il est à remarquer que cette méthode est inefficace si le concepteur d'un virus change simplement le code de reconnaissance du virus. Cette méthode peut être utile si une organisation a subi des infections par un même virus. Par exemple, après avoir nettoyé trois ou quatre infections d'un virus particulier sur un réseau ou un PC, l'inoculation est à considérer alors comme une mesure désespérée.

2.6 Quelques tendances dans le développement de virus

Durant les dernières années, on a pu observer de nombreuses activités dans le camp des auteurs de virus. Beaucoup de nouvelles techniques ont été inventées et expérimentées. Certaines se sont couronnées de succès, d'autres moins. Quelques unes vont se généraliser sans doute à l'avenir.

La tendance principale dans l'écriture de virus qu'on observe actuellement est la production croissante de nouveaux virus. Il y en a de plus en plus - environ 1000 par an! Cela crée déjà des problèmes pour les sociétés qui développent des anti-virus. D'abord, c'est quasi impossible de commencer à zéro pour quelqu'un qui décide de se lancer dans ce commerce : il y a actuellement environ 9000 virus connus et le nombre ne cesse d'augmenter. Pour les autres, les scanners existants sont de plus en plus difficiles à maintenir - vu la quantité de signatures de reconnaissance à introduire dans le programme.

Voici d'autres possibilités permettant un développement plus facile de virus :

- Il existe des «Authoring packages», c'est à dire des programmes qui aident au développement de virus. Ils rendent les dernières techniques de rédaction de virus accessibles au grand public - cela risque d'accélérer encore le problème énoncé plus haut. Autre danger : ces paquets génèrent du code source et permettent l'étude de certaines techniques que des gens pourraient utiliser dans leurs propres virus. En plus, ces outils sont généralement écrits dans des langages de haut niveau, et le code machine du virus ainsi généré est dès lors difficile à comprendre.
- Une approche corrélée est celle des «Virus mutators». L'idée est de créer un programme qui prend un programme existant (par exemple, un virus) et le modifie en créant un programme équivalent. Ce « mutateur » (programme qui crée donc les mutations de programmes) appliquerait cette méthode au programme entier. Supposons qu'un auteur de virus utilise ce « mutateur » et l'applique à une collection de virus. Le résultat sera des centaines de nouveaux virus générés. Dès lors, on peut comprendre qu'un tel programme rendrait rapidement obsolète tout scanner.
- De nouveaux types de virus apparaissent : des virus qui se spécialisent dans les réseaux locaux en exploitant certaines interruptions non documentées dans le gestionnaire de réseau de Novell par exemple, ou bien en profitant de bugs dans des algorithmes de chiffrement pour voler des mots de passe par exemple.
- Un tout nouveau danger est en train d'apparaître : des virus qui se répandent via Internet et spécialement via le nouveau langage universel JAVA. Personne ne peut encore dire à quoi ce danger va concrètement aboutir. Tout dépendra de la puissance des fonctions qui seront implémentées dans ce langage et du niveau de sécurité qu'on imposera au système.

3. Audit, V-IDES et Pandora

3.1 Introduction

Dans ce chapitre, nous expliquerons d'abord le souci de sécurité et les grands principes de l'auditing dans les systèmes informatiques, puis nous décrirons un modèle d'automatisation d'évaluation de traces d'exécution - le modèle IDES et évoquerons brièvement la spécialisation du modèle IDES dans le cadre des virus informatiques - Virus-IDES et son implémentation dans le monde des PC IBM-Compatibles, appelée PANDORA.

3.2 L'audit

3.2.1 Définition

L'audit est un mécanisme de sécurité utilisé dans les systèmes informatiques. Le combat pour la sécurité d'un système d'exploitation doit être mené sur plusieurs fronts à la fois. Un système multiutilisateur doit protéger les fichiers, la mémoire et les autres ressources de chaque utilisateur. Il doit aussi protéger les données, les fichiers et la mémoire du système d'exploitation contre les programmes utilisateurs. Il doit également surveiller les tentatives de contournement de ces fonctions de sécurité, et ainsi de suite.

Dans un système sécurisé, le processus d'auditing consiste en l'enregistrement, l'examen et l'analyse de certaines ou de toutes les activités relevantes pour la sécurité dans le système. On parle d'activités relevantes pour la sécurité du système dans le sens d'événements qui essaient de changer l'état de sécurité du système (par exemple, trop de tentatives de login, ...)
Aux USA, le NCSA (National Computer Security Center) a établi une liste de critères pour évaluer l'efficacité de contrôles de sécurité dans des systèmes ADP (*Automatic Data Processing*), i.e. des systèmes de traitement automatique de données. D'après le guide du NCSA, un système ADP est un « ensemble de software, hardware et firmware informatiques configurés dans le but de classer, trier, estimer, calculer, résumer, transmettre et recevoir, emmagasiner et retrouver des données avec un minimum d'intervention humaine. »

Ces caractéristiques sont regroupées en sept niveaux de sécurité; catégorisées en 4 divisions D - C - B - A, ordonnées de manière hiérarchisée avec la note A réservée aux systèmes qui incorporent le niveau le plus élevé d'assurance. Endéans les divisions C et B, il y a un certain nombre de subdivisions appelées classes qui sont également ordonnées.

La particularité des critères C2 à A1 (un seul pour la division A) est l'obligation d'enregistrement de toutes les actions d'un utilisateur par moyen d'audit ou, plus généralement, toutes les activités qui se produisent dans le système.

3.2.2 Principes d'auditing

Les audit-trails sont le compte-rendu d'événements qui ont eu lieu dans le système, qui ont été enregistrés pour pouvoir garder une trace de leur exécution. Par exemple, ces traces d'exécution peuvent être utilisées pour détecter une pénétration dans un système informatique et découvrir tous les abus.

Un audit-trail est un fichier séquentiel composé d'enregistrements de taille variable. L'administrateur décide quels événements il faut logger en vue de pouvoir retracer l'évolution d'un incident ultérieurement pour en trouver les causes. Cela peut se faire en utilisant deux méthodes de sélection : la pré-sélection et la post-sélection.

Puisqu'il est impossible de tracer tous les événements système, les administrateurs procèdent la plupart du temps à une pré-sélection. « La pré-sélection est utilisée pour contrôler de manière sélective la génération d'enregistrements audit. Cela permet à certains utilisateurs et événements de générer des enregistrements audit tandis que d'autres sont écartés. Le résultat est un audit-trail plus compact avec moins de détail que si un audit complet était effectué »[3].
Un désavantage de la pré-sélection est qu'il est très dur de prédire quels événements pourraient devenir relevants pour la sécurité à une date ultérieure. Il y a toujours la possibilité que des événements non pré-sélectionnés deviennent un jour relevants pour la sécurité, et la perte potentielle due au fait de ne pas avoir audité ces événements serait impossible à déterminer.
L'avantage de la pré-sélection pourrait être une meilleure performance résultant du fait que tous les événements du système n'ont pas été audités.

« La post-sélection est l'utilisation sélective de données audit collectées. La post-sélection inclut la collection des données audit pour tous les événements et utilisateurs de telle manière que l'audit-trail soit aussi complet que possible »[4] .Dans la post-sélection, on décide a posteriori de quels événements il faut tenir compte par la suite, ayant à sa disposition des traces de tout ce qui a été jugé nécessaire lors de la pré-sélection. On utilise en quelque sorte un filtre sur un fichier qui contient les traces d'un nombre important d'événements qui se sont produits dans le système.
L'avantage principal de la post-sélection est que l'information qui pourrait se révéler utile dans le futur est déjà enregistrée sur un audit-trail et peut être cherchée à tout moment.
Le désavantage de la post-sélection pourrait être une performance dégradée par l'écriture et l'emmagasinage de ce qui pourrait devenir un très grand audit-trail.

Il est bien sûr sous-entendu que les fichiers d'audit doivent être protégés de manière très stricte contre toute modification non autorisée. De plus, seul le responsable d'audit doit être en mesure de fixer les événements à auditer et à modifier les choix.

On pourrait imaginer une pré-sélection seule, ou bien seulement une post-sélection, mais seul leur complémentarité permet des performances optimales. Illustrons cela : lors de la pré-sélection, on délimite les champs de supervision à une quantité supportable d'événements ; ensuite, la post-sélection permet de se focaliser sur certains aspects en extrayant les informations pertinentes pour un type d'investigation, ce qui accélère bien sûr les traitements.

3.2.3 Objectifs

Le mécanisme d'audit a cinq grandes missions de sécurité:

1. retracer des schémas d'accès à des objets individuels, livrer l'historique d'accès de processus et d'individus, et permettre un examen de l'efficacité des moyens de protection.
2. rendre possible la détection de tentatives répétitives d'utilisateurs et d'extérieurs à contourner les mécanismes de protection.
3. découvrir l'abus de privilèges pour nuire au système.

[3] D'après SINIXS V5.22 Security Features Administrator's Guide
[4] D'après SINIXS V5.22 Security Features Administrator's Guide

4. agir de manière dissuasive envers chacun qui se sent tenté par une infiltration dans le système. Pour cela, chaque agresseur potentiel doit être mis au courant de l'existence de mécanismes d'audit et de son utilisation pour détecter toute tentative de contournement du système de protection.
5. limiter les dégâts même si quelqu'un réussit à s'infiltrer. En effet, l'action ne passera pas inaperçue car tout est enregistré et elle sera donc découverte. On sait retrouver l'auteur d'un incident grâce à l'audit-trail.

3.2.4 Exemple

A titre d'exemple, citons les caractéristiques qui doivent être implémentées pour un niveau de sécurité de la classe C2 :

- *un service d'ouverture de session sécurisé* demandant l'identification des utilisateurs à l'aide d'un identificateur de session unique et d'un mot de passe, avant qu'ils ne puissent accéder au système,
- *un contrôle d'accès discrétionnaire* permettant au propriétaire d'une ressource de déterminer qui peut y accéder et ce qui peut en être fait ; ceci en accordant des droits d'accès à un utilisateur ou un groupe d'utilisateurs,
- *l'audit* permettant de détecter et d'enregistrer des événements importants concernant la sécurité, ou toute tentative de créer, d'accéder ou de supprimer des ressources du système. Il utilise les identificateurs d'ouverture de session pour enregistrer l'identité de l'utilisateur ayant effectué l'action,
- *la protection de la mémoire*, interdisant tout d'abord à un processus d'accéder aux ressources mémoires d'un autre processus et empêchant une lecture de cette mémoire après sa libération. La mémoire est donc réinitialisée avant une nouvelle mise à disposition.

Les événements qui doivent être audités pour respecter le critère C2 sont :

- l'utilisation de mécanismes d'identification et d'authentification,
- l'introduction d'objets dans l'espace d'adressage de l'utilisateur,
- l'effacement d'objets de l'espace d'adressage de l'utilisateur,
- les actions entreprises par l'opérateur et les administrateurs système et/ou administrateurs de sécurité,
- et évidemment, les autres événements considérés comme relevants pour la sécurité.

Les informations à enregistrer dans la trace d'exécution sont :

- la date et l'heure de l'événement,
- l'identifiant unique de celui qui a généré l'événement,
- le type d'événement,
- l'origine de la requête (ex. ID du terminal) pour des événements d'identification et d'authentification,
- les noms des objets introduits, accédés ou effacés de l'espace utilisateur,
- la description des modifications faites par l'administrateur du système aux bases de données de sécurité du système ou de l'utilisateur.

3.3 V-IDES

Les audit-trails sont utilisés depuis longtemps pour détecter des activités illicites.
Analyser cette quantité d'informations à la main est impossible. Les premiers modèles d'analyse automatique sont apparus début des années 80. Au milieu des années 80, le modèle IDES (INTRUSION DETECTION EXPERT SYSTEM) fut développé par Dorothy Denning.

Le principe en était la détection des attaques au système par un comportement anormal du sujet. Ce comportement pouvait être analysé moyennant l'audit-trail.
Au début, ces tests portaient sur le détournement d'un profil considéré comme normal, i.e. veiller à ce que le comportement se déroule dans des limites standards. Une certaine tolérance au dépassement était accordée avant qu'un comportement ne soit pris pour une atteinte à la sécurité.

Cette analyse statistique est en principe capable de détecter des attaques inconnues. Mais elle est impuissante contre des « insiders » qui abusent de leurs pouvoirs pour commettre des actes prohibés. Ces actes peuvent passer inaperçus car ils restent dans les bornes de tolérance. Prenons un exemple: si un utilisateur a réussi à s'approprier le mot de passe de l'administrateur, il peut faire beaucoup plus que ce que ses droits ne lui auraient permis, par exemple, effacer ou modifier des fichiers du système.

Plus tard, le modèle IDES fut élargi par une composante basée sur des règles. Ces règles sont principalement employées pour trouver des attaques connues non détectables par des analyses statistiques. Elles reconnaissent des suites d'actions qui sont connues comme stratégie d'attaque. Ces règles n'ont plus besoin d'informations sur les sujets. Elles traduisent les scénarios qui sont considérés comme violation du système. Donc, ce qui sera finalement détecté, n'est pas ce qui est dangereux pour un système, mais ce qui est considéré comme dangereux! On peut en déduire que les règles doivent constamment évoluer ou être affinées pour tenir compte du savoir actuel.

Pour appliquer ces idées dans le monde des virus informatiques, le système IDES fut adapté dans ce sens par le Dr. Fischer-Hübner à l'Université de Hambourg, Département Informatique. Son nom est VIDES (Virus-IDES) et ses objectifs sont de :

- reconnaître avec grande certitude un fichier infecté comme atteint par un virus,
- trouver le plus de renseignements possibles sur le virus détecté, entre autre ses mécanismes d'infection.

Il devrait être possible de reconnaître tous les virus utilisant les schémas classiques d'infection.
Le système complet consiste en un système d'audit et un système expert .
L'audit collecte des traces d'exécution d'une session où le virus est exécuté. Ces traces sont évaluées plus tard dans le système expert.

3.4 Auditing sous PC-DOS et PANDORA

3.4.1 Introduction

La condition préalable à l'utilisation d'un système de détection d'intrusion par analyse dynamique de traces d'exécution est un système d'audit qui collecte de manière sécurisée les données correspondant à l'activité du système. De plus, l'intégrité du système de détection ne doit pas être compromise, ce qui signifie que la recherche, l'analyse et l'archivage des données audit doivent être sécurisés contre la corruption par les virus.
Seulement, le DOS ne fournit pas un tel service et ne rend pas l'implémentation d'un tel service facile. Son manque total de mécanismes de sécurité signifie que la collection des données peut être falsifiée. Même si la collection peut être sécurisée, la donnée est ouverte à la manipulation si elle est emmagasinée sur la même machine.
Donc il y a obligation de créer un tel système et de l'écrire complètement. Celui-ci et les traces d'exécution devront être protégés contre toute manipulation par d'autres programmes qui tournent.
Pour le prototype de VIDES et pour mettre en œuvre les idées du concept de Dr. Simone Fischer-Hübner, plusieurs possibilités ont été explorées par Morton Swimmer et elle-même. Voici historiquement le compte-rendu des quelques alternatives tentées pour aboutir finalement à PANDORA.

3.4.2 Alternatives

3.4.2.1 Interruptions DOS

Tous les services DOS sont fournis aux programmes via des interruptions. Le service demandé et ses paramètres sont entrés dans des registres mémoire. Quand le service est fini, il rend le contrôle au programme appelant et fournit ses résultats dans des registres.
La toute première implémentation d'un système audit était un filtre qui était placé devant les services DOS et enregistrait tous les appels aux fonctions DOS.
Le plus gros problème était l'insécurité des données d'audit. Elles se trouvaient en effet sur la machine où avait lieu l'analyse, et étaient donc exposées au même risque de manipulation par le virus que toutes les autres données, idem pour le filtre qui était conçu comme un programme DOS normal.

Cette première implémentation démontra :

- que le système d'audit ne tournait pas de manière fiable et pouvait être détourné par des virus *tunneling*,
- mais que les virus pouvaient par contre être trouvés en utilisant une technique de détection d'intrusion.

3.4.2.2 Machine virtuelle 8086

La première machine à fournir le mode virtuel 8086 fut l'Intel 386.
Des systèmes d'exploitation tournant en mode protégé peuvent créer beaucoup de machines 8086 virtuelles dans lesquelles des tâches peuvent s'exécuter en étant totalement isolées l'une de l'autre et du système d'exploitation. Chaque tâche « voit » seulement son propre environnement. Le système d'exploitation OS/2 par exemple utilise ce système pour fournir un environnement DOS complet pour les programmes DOS. Tous les appels au DOS sont redirigés au système d'exploitation maître pour leurs traitements.

Ce mécanisme peut également être utilisé pour surveiller l'activité dans la session DOS. Puisque toutes les interruptions sont redirigées vers le système d'exploitation maître, ce dernier peut de manière sécurisée et discrète enregistrer l'activité.
Malheureusement durant les tests, beaucoup de virus tournant sur une fenêtre DOS ont atteint des parties vitales du système. Par exemple, dans le cas d'OS/2, le problème était que les fichiers OS/2 étaient directement manipulables à l'intérieur de la session DOS et certains virus avaient réussi à les modifier.

Il fallait donc chercher une autre méthode plus sure pour protéger le système d'exploitation de l'exécution du virus.

3.4.2.3 Support hardware

Des systèmes hardware de traçage (par exemple Periscope IV) peuvent être utilisés pour surveiller des événements système en temps réel. Cela est réalisé concrètement par une carte placée entre le CPU et la carte-mère et sur laquelle on peut définir des points d'arrêt pour différents types d'événements qui passent sur le bus du PC. La carte est connectée à une carte réceptrice dans un autre PC qui est utilisé pour contrôler cette session de traçage.
La surveillance est totalement discrète, i.e. que le programme ne remarquera pas une différence dans son exécution avec ou sans le traçage. En effet, quand un événement est déclenché, le PC est arrêté tandis que le PC de contrôle traite la donnée. Si le PC de contrôle est assez rapide, le délai de temps de traitement est négligeable.
Malheureusement, une solution hardware est pratiquement impossible à mettre en œuvre car pour chaque système d'exploitation où l'on effectue des tests, il faut deux machines pour le traçage. Cela devient alors vite très coûteux.

3.4.2.4 L'émulation 8086

La solution qui a été finalement retenue est l'émulation software du processeur 8086. Un émulateur est un programme qui accepte l'ensemble complet des instructions d'un processeur en entrée et interprète le code binaire comme le processeur original le ferait. Tous les autres éléments de la machine doivent être implémentés ou émulés comme, par exemple, les différents ports.
L'utilisation d'une émulation nous offre tous les avantages de la solution hardware plus la possibilité de manipuler tous les événements du programme s'exécutant dans l'émulation.
Le problème de la protection du système d'audit vis-à-vis de la session DOS est aussi résolu car le virus n'a pas du tout accès à la machine maître. C'est parce que la mémoire de la machine est entièrement contrôlée par l'émulation et les accès aux fichiers sont dirigés vers un disque virtuel sous la forme de fichiers images. Ces fichiers contiennent la même structure binaire qu'un disque (secteur de démarrage, division en secteurs, répertoires ...). Dans notre cas, Morton Swimmer a utilisé un émulateur sous UNIX et des fichiers images de 1.44 MB pour émuler des disquettes DOS. Chaque fichier image contient une version DOS différente, permettant ainsi de tester les virus dans différentes version de DOS.
Puisque les audit-trail sont sauvés comme fichiers binaires à l'extérieur (dans le système de fichiers UNIX), ils sont complètement à l'abri.
Chaque code opérationnel ou appel système peut ainsi être mémorisé.

Tout ceci est bien intéressant, mais l'émulation a aussi des limites.
D'abord, le problème majeur de l'utilisation d'une émulation est sa lenteur. Même sur des plates-formes rapides, la vitesse d'exécution est seulement un peu plus rapide qu'un PC/XT original (Intel 8086). Ensuite, la complexité de couvrir le plus de types d'instructions, i.e. de supporter aussi les processeurs avancés est donc très difficile à résoudre.
Enfin, le grand travail d'écrire une émulation est un autre problème.

3.4.3 Le format des données d'activité et **PANDORA**

La dernière alternative présentée auparavant a été celle retenue pour PANDORA, système que nous avons eu à notre disposition pendant le stage pour la génération des fichiers d'audit.

La disponibilité du code source de l'émulateur PCEmu de David Hedley (Université de Bristol UK) avait permis à Morton Swimmer de l'étendre avec des modules d'audit.
Durant l'exécution d'un virus dans l'émulateur, le système d'audit intercepte pratiquement toutes les interruptions 0x21 DOS. En effet, c'est principalement cette interruption qui est utilisée par les programmes pour demander des services DOS. Or, comme un virus a tout intérêt à rester le plus général possible pour pouvoir infecter le plus de fichiers, il doit utiliser les moyens standards pour ses requêtes DOS. C'est pour cela qu'on audite cette interruption 21. Cette interruption est décomposée en un nombre important de fonctions DOS.
Pandora capte tous les appels utilisant le mécanisme d'interruptions, mais seulement les fonctions DOS les plus couramment utilisées et les plus intéressantes pour l'analyse ultérieure sont auditées et placées dans la trace d'exécution.
Lors d'un appel d'interruption, quatre registres généraux (de AX à DX) et 6 autres registres peuvent changer de valeur, mais seulement certains sont vraiment modifiés. D'où l'idée, pour l'alléger, de ne stocker dans le fichier audit-trail que les registres effectivement changés lors de l'appel.

Parmi ces fonctions auditées, on trouve

- Open : Ouverture d'un fichier et assignation d'un handle (numéro d'identification)
- Close : Fermeture d'un fichier par son handle
- Create : Création d'un nouveau fichier
- Read : Lecture dans un fichier préalablement ouvert
- Write : Ecriture dans un fichier ouvert
- Unlink : Effacement d'un fichier sur disque
- Lseek : Déplacement de la tête de lecture/écriture dans un fichier
- GetDateTime : Chargement de la date et de l'heure du fichier concerné
- SetDateTime : Modification de la date et de l'heure du fichier concerné
- Rename : Renommage d'un fichier
- ....

Les champs des enregistrements audit collectés par l'émulateur PC ont la signification suivante :

- *codesegment* est l'adresse en mémoire du programme duquel vient l'appel, utile pour discerner la portion mémoire qu'occupe le virus dans son programme hôte,
- *function number* est le numéro de la fonction DOS appelée par le programme,
- *arg(...)* est la liste des valeurs de registres mémoire utilisés dans l'appel de la fonction DOS,
- *ret(...)* est la liste des valeurs de registres mémoire retournés par la fonction appelée, à l'intérieur de laquelle se trouve souvent le CF (*Carry Flag*) qui renseigne sur le fait que la fonction a réussi ou échoué.
- *RecType* est le type de l'enregistrement,
- *StartTime* et *EndTime* sont les timestamps respectifs de l'action de début et de fin de la fonction DOS.

La description précise des différents champs des enregistrements audit peut être trouvée à l'annexe 3.

Un exemple d'une partie d'un audit-trail est donné :

...

```
<CS=3911 Type=0 Fn=30 arg() ret(AX=5) >
<CS=3911 Type=0 Fn=29 arg() ret(BX=128 ES=3911) >
<CS=3911 Type=0 Fn=64 arg(AL=61 CL=3 str1=*.COM) ret(AL=0 CF=0) >
<CS=3911 Type=0 Fn=51 arg(AL=0 str1=COMMAND.COM) ret(AL=0 CX=32 CF=0) >
<CS=3911 Type=0 Fn=51 arg(AL=1 str1=COMMAND.COM) ret(AL=0 CX=32 CF=0) >
<CS=3911 Type=0 Fn=45 arg(AL=2 CL=32 str1=COMMAND.COM) ret(AL=0 AX=5 CF=0) >
<CS=3911 Type=0 Fn=73 arg(BX=5) ret(CX=10241 DX=6206 CF=0) >
<CS=3911 Type=0 Fn=27 arg() ret(CX=5121 DX=8032) >
<CS=3911 Type=0 Fn=47 arg(BX=5 CX=3 DX=828 DS=3911) ret(AX=3 CF=0) >
<CS=3911 Type=0 Fn=50 arg(AL=2 BX=5 CX=0 DX=0) ret(AL=0 AX=50031 DX=0 CF=0) >
<CS=3911 Type=0 Fn=48 arg(BX=5 CX=648 DX=313 DS=3911) ret(AX=648 CF=0) >
<CS=3911 Type=0 Fn=50 arg(AL=0 BX=5 CX=0 DX=0) ret(AL=0 AX=0 DX=0 CF=0) >
<CS=3911 Type=0 Fn=48 arg(BX=5 CX=3 DX=831 DS=3911) ret(AX=3 CF=0) >
<CS=3911 Type=0 Fn=74 arg(BX=5 CX=10271 DX=6206) ret(CF=0) >
<CS=3911 Type=0 Fn=46 arg(BX=5) ret(CF=0) >
<CS=3911 Type=0 Fn=51 arg(AL=1 str1=COMMAND.COM) ret(AL=0 CX=32 CF=0) >
```

...

Cet exemple est tiré de l'audit-trail au format NADF (voir seciton 4.2.) de l'exécution du virus Vienna mis sous la forme lisible pour l'homme.

4. A.S.A.X.

La figure suivante montre la vue d'ensemble conceptuelle d'ASAX. Les concepts qui y sont évoqués sont l'objet d'un développement plus important dans les sous-sections suivantes.

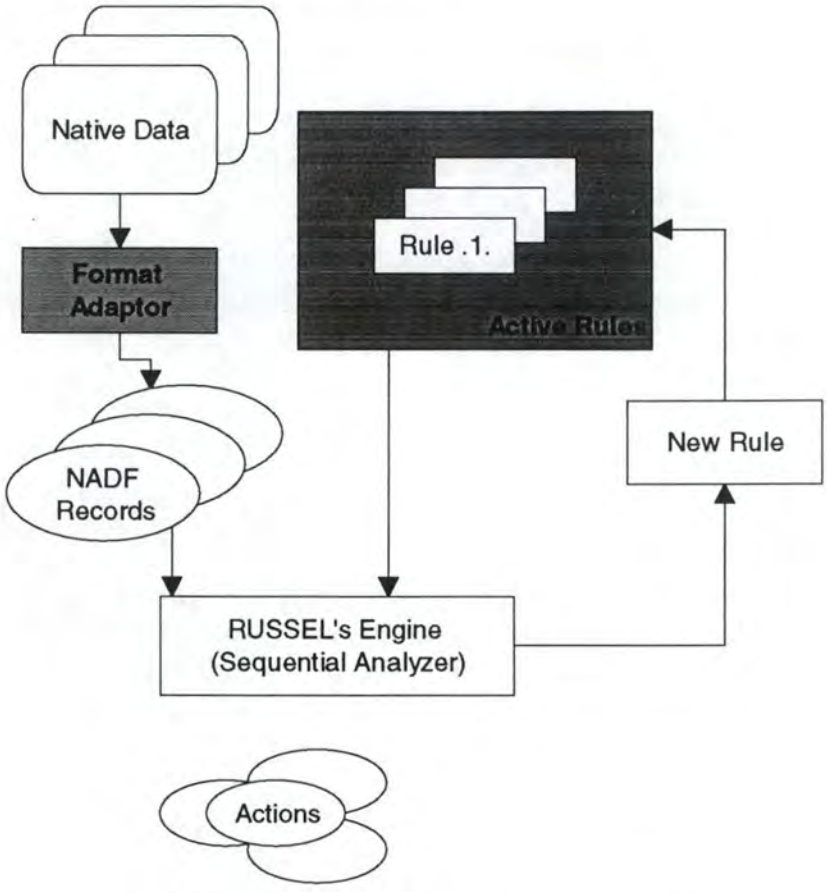


Fig. 4.1 : ASAX Conceptual Overview

4.1 Caractéristiques

A.S.A.X. : **A**dvanced **S**ecurity **A**udit Trail Analysis on uni**X** est né d'un projet entre Siemens Nixdorf à Rhisnes et l'Institut d'Informatique des Facultés Universitaires Notre Dame de la Paix à Namur. Son but était la création d'un logiciel permettant l'analyse d'audit-trail venant de tous horizons.

Les caractéristiques principales d'ASAX sont :

- l'universalité,
- la puissance,
- l'efficacité,
- la portabilité,
- l'extensibilité.

4.1.1 Universalité

ASAX est un produit universel dans le sens qu'il peut analyser n'importe quel type de fichier séquentiel. Cet objectif est atteint en traduisant des fichiers au format natif - c'est-à-dire au format fourni par le programme d'audit - dans un format générique appelé *Normalized Security Audit Data Format* (NADF).
Le format NADF est extrêmement simple et flexible, ainsi, n'importe quel type de format d'audit peut être traduit théoriquement dans ce format générique.
De plus, ASAX fournit une grande quantité d'adaptateurs de format (*format adaptors*) qui convertissent des données existantes dans le format NADF.
Ce choix de prendre un format générique élimine ainsi le besoin de devoir adapter ASAX à toutes les sources possibles de données.
La section 4.2. de ce chapitre développe le format NADF et les adaptateurs de format.

4.1.2 Puissance

Pour réaliser cette caractéristique, ASAX utilise un langage basé sur les règles (*rules*) et conçu spécifiquement pour l'analyse de fichiers séquentiels énormes : RUSSEL (**RU**le-ba**S**ed **S**equential **E**valuation **L**anguage).
Une règle est une sorte de procédure permettant de retrouver des échantillons particuliers d'enregistrements dans des fichiers séquentiels et de déclencher des actions comme l'activation d'alarmes, l'envoi de messages, ...
Une distinction particulière sépare cependant une procédure d'une règle : chaque règle peut déclencher de nouvelles règles.
Le langage RUSSEL montre sa puissance en fournissant des structures de contrôle classique ainsi qu'un mécanisme de déclenchement basé sur les règles (*rule-based triggering mechanism*) permettant d'exprimer les requêtes les plus compliquées.
Ce langage sera plus amplement présenté dans la section 4.3. de ce chapitre.

4.1.3 Efficacité

Cette caractéristique est importante car l'analyse porte souvent sur des fichiers énormes. Celle-ci est réalisée grâce à deux principes-clés sous-jacents au design même d'ASAX :

1. Grâce au design du langage RUSSEL, l'analyse du fichier audit est faite en une et une seule passe. L'information que l'on doit garder sur le passé, i.e. à propos des enregistrements déjà analysés est soit passée comme paramètre dans les règles suivantes, soit en conservant l'information nécessaire dans des tables associatives comme nous le verrons dans le chapitre 5.
2. ASAX optimise autant que possible les démarches répétitives par des techniques d'implémentation efficaces et sophistiquées.

4.1.4 Portabilité

ASAX a été programmé de telle manière qu'il est facilement portable sur n'importe quel type de machine tournant sous n'importe quel type de système d'exploitation (où il existe au moins un compilateur C). Il a été ainsi déjà porté sur les systèmes d'exploitation DOS, UNIX, SINIX et tourne théoriquement avec succès sur tout type de machine.

4.1.5 Extensibilité

Le langage RUSSEL supporte une large variété de fonctions s'étalant de la sélection simple ou multiple d'enregistrements et incluant des fonctionnalités de comptes-rendus, de statistiques et d'alarmes.
ASAX est d'ailleurs fourni avec un ensemble de programmes supportant ces fonctionnalités mais d'autres sont faciles et rapides à implémenter.

De plus, RUSSEL a été conçu de manière à offrir une interface claire avec le langage C. Grâce à cela, les fonctionnalités d'ASAX peuvent être facilement et rapidement étendues.
Dans le projet de notre stage, nous avons développé une extension en C de tables associatives qui sera développée dans le chapitre 5.

4.2 Le format NADF et les adaptateurs de format

Le kernel ASAX ne reconnaît que les enregistrements sous le format NADF. Pour être donc capable d'analyser les fichiers audit, il faut d'abord passer par un traitement sur les données originales (qui sont sous n'importe quel format). Pour cette traduction, on utilise un adaptateur de format (*format adaptor*).
De plus, tous les types de données qui peuvent être trouvés dans le fichier transformé doivent être décrits dans un fichier de description des données (*data description file*).
En résumé, pour préparer le fichier de données qui doit être analysé, il faut fournir :

1. un adaptateur de format,
2. un fichier de description des données.

4.2.1 Le format NADF

Le format NADF (**N**ormalized Security **A**udit **D**ata **F**ormat) a été spécialement conçu pour les objectifs que recherchait ASAX.
Dans ce format, chaque enregistrement consiste en :

- la longueur réelle (sur 4 bytes) de l'enregistrement i.e. la longueur du contenu de l'enregistrement et les 4 bytes du champ longueur,
- le contenu de l'enregistrement lui-même.

Le contenu de l'enregistrement consiste en une séquence de champs. Un champ est représenté par :

- son identifiant (entier de 2 bytes)
- sa longueur (entier de 2 bytes)
- sa valeur (type fixé par l'identifiant)

Chaque champ a son nom lié à l'identifiant par une relation 1-1.
La longueur du champ est en fait la longueur de sa valeur.

Real length of record		
id_1	$lgth_1$	$value_1$
...	...	...
id_i	$lgth_i$	$value_i$
...	...	...
id_n	$lgth_n$	$value_n$

Fig. 4.2 : Représentation d'un enregistrement au format NADF

La séquence de champs est triée selon le champ identifiant pour augmenter la performance du traitement de l'enregistrement au format NADF.
Le codage exact de ces champs n'est pas utile car il dépend de la machine sur laquelle l'analyse est exécutée. On peut seulement dire que id_i et $lgth_i$ sont entiers mais leurs représentations sont dépendantes de la machine. Dans RUSSEL, $value_i$ est traitée comme une chaîne de bytes.

4.2.2 Les adaptateurs de format

Les adaptateurs de format s'occupent de la traduction des enregistrements en entrée sous le format NADF.
Il en existe deux formes :

- sous la forme de programmes localisés hors du kernel ASAX
- ou sous la forme de trois routines I/O.

Les premiers attendent en entrée l'ensemble des données à transformer et génèrent le fichier NADF correspondant. L'avantage de cette forme est de transformer le fichier d'entrée une et une seule fois et de garder le résultat de la transformation dans un fichier NADF qui peut être alors directement analysé.

Les trois routines I/O quant à elles, supportent l'ouverture (routine *vopen*), la fermeture (*vclose*) du fichier à analyser et la lecture (*vread*) d'un enregistrement avec la transformation de cet enregistrement sous le format NADF incluse. L'avantage de ce type d'adaptateur est qu'il évite la génération de fichiers NADF. La transformation est ainsi intégrée à l'analyse, ce qui signifie qu'elle est réalisée progressivement, enregistrement par enregistrement, avec l'opération de lecture. Cet adaptateur est un prérequis pour l'analyse on-line.

Une donnée audit est simplement transformée en ses identifiant, longueur et valeur.
Il faut savoir que la valeur de l'enregistrement au format NADF conserve la forme codée de l'enregistrement original (pour des raisons de sécurité, d'impossibilité de décodage,...). La tâche de l'adaptateur de format ne se limite donc pas à une traduction pure et simple, celui-ci s'occupe également de conserver les règles de *mapping* entre la représentation interne et externe de la donnée audit.

4.2.3 Les fichiers de description de données

Un fichier de description de données (*data description file*) est un fichier qui contient la description de l'information qui peut être trouvée dans le fichier à analyser (sous sa forme NADF évidemment).

C'est un fichier texte séquentiel contenant un ensemble de description de données précédé de quelques lignes d'en-tête.
Optionnelles, les six lignes d'en-tête fournissent de l'information générale.
Une description de données représente la description d'un champ en 5 lignes :

1. l'identifiant (qui sera l'identifiant sous le format NADF) ,
2. le type d'origine,
3. le type cible,
4. le nom (qui sera la référence à cette donnée dans un programme d'analyse),
5. la description sémantique (optionnelle).

A l'annexe 3 se trouve le fichier de description de données que nous avons utilisé à notre stage. Il représente la description des différents champs (qui peuvent être utiles dans une analyse future au moment d'une interruption DOS) que contient chaque enregistrement contenu dans le fichier NADF obtenu après lancement d'un virus quelconque sous DOS.

4.3 Le langage RUSSEL

RUSSEL (**RU**le-ba**S**ed **S**equence **E**valuation **L**anguage) est un langage spécifiquement façonné pour le problème de la recherche d'échantillons arbitraires d'enregistrements dans des fichiers séquentiels.
Un programme RUSSEL consiste simplement en un ensemble de déclarations de règles. Ces déclarations sont composées d'un nom de règle, d'une liste de paramètres formels et de variables locales, et d'une partie que nous allons appeler partie action (qui est en fait un ensemble d'actions). Ces actions peuvent être conditionnelles, répétitives et composées ; et le langage inclut également les actions primitives d'assignation, d'appel de routine externe et de déclenchement de règles. De plus, RUSSEL supporte les modules partageant des variables et des déclarations de règles exportées.

4.3.1 Types de données

Seulement deux types de données de base sont considérés :

- les chaînes de bytes (*string of bytes*)
- les entiers (*integer*)

Les types gardés sont aussi simples que possible pour assurer la compatibilité avec une grande variété de formats d'audit. En théorie, toute donnée peut être représentée par une chaîne de bytes, c'est d'ailleurs le type de donnée le plus utilisé en RUSSEL. Les entiers sont introduits pour prendre l'avantage des opérations arithmétiques pour spécifier des opérations de sélection complexes sur les enregistrements.

4.3.2 Syntaxe

4.3.2.1 Eléments lexicaux

Un programme ASAX est une séquence finie d'éléments lexicaux. On peut trouver la définition BNF des éléments lexicaux en annexe 2.1.

4.3.2.2 Syntaxe abstraite

La définition BNF de la syntaxe abstraite du langage RUSSEL se trouve en annexe 2.2.
L'exactitude syntaxique complète d'une expression, condition ou construction dépend d'une table d'identifiants rassemblant les noms des paramètres, des variables et des champs. Cette table est simplement appelée « table des identifiants » (*identifier table*).

4.3.2.3 Syntaxe concrète

La définition BNF de la syntaxe concrète du langage RUSSEL se trouve en annexe 2.3.
Cette syntaxe concrète permet de résoudre les ambiguïtés provenant de problèmes sur les expressions par exemple.
De la définition BNF, on peut tirer implicitement un tableau de priorité des opérateurs.

Priority	*Operators*
1	* , mod, div
2	+,-
3	<,>,≥,≤,=,≠
4	not, present
5	and
6	or

Fig. 4.8 : Tableau de priorité des opérateurs

Ainsi, une telle expression :

$$a + b * c < c + d$$

est interprétée de la manière suivante :

$$(a + (b * c)) < (c + d)$$

4.3.2.4 Règles syntaxiques additionnelles

1. Une opération arithmétique s'applique seulement aux expressions entières.
2. Tout identifiant doit apparaître dans la table des identifiants.

4.3.3 Sémantique

Le but de cette section est de donner une signification à chacune des constructions du langage RUSSEL. Mais il faut d'abord introduire deux concepts de base :

- l'enregistrement actuel (*current record*) est l'enregistrement traité à un moment donné dans l'analyse du fichier audit.
- l'environnement actuel (*current environment*) est l'ensemble d'informations dont on a besoin pour décrire précisément comment les instructions du programme ASAX sont exécutées. Un environnement actuel comprend :
  - l'enregistrement actuel,
  - l'environnement local : un ensemble de variables,
  - DStrig : l'ensemble des règles *actives* ou *déclenchées* (*triggered*),
  - DSnext : l'ensemble des règles à déclencher pour l'enregistrement suivant.
  - DScompl : l'ensemble des règles d'achèvement (*completion rules*), i.e. l'ensemble des règles à déclencher quand le traitement de l'entièreté du fichier a été réalisé.

4.3.3.1 Sémantique opérationnelle

On peut décrire la sémantique opérationnelle de RUSSEL de cette manière :

1. Les enregistrements sont analysés séquentiellement. L'analyse de l'enregistrement actuel consiste en l'exécution de toutes les règles actives contenues dans DStrig. L'exécution d'une règle active peut déclencher de nouvelles règles, déclencher des alarmes, écrire des messages de compte-rendu ou modifier des variables globales, etc.
2. Le déclenchement de règles (*rule triggering*) est un mécanisme spécial par lequel une règle est rendue active soit pour l'enregistrement actuel soit pour le suivant soit à la fin de l'analyse du fichier. Ce mécanisme est expliqué dans la section 4.3.3.3.
3. Quand toutes les règles actives pour l'enregistrement actuel ont été exécutées, l'enregistrement suivant est lu et les règles déclenchées pour celui-ci à l'étape précédente dans l'ensemble DSnext sont exécutées à leur tour.
4. Pour initialiser l'analyse, un ensemble de règles initiales (*init rules*) sont rendues actives pour le premier enregistrement par une procédure spéciale appelée **init_action**.

Un exemple sera développé dans la section 4.3.7. pour mieux expliquer la sémantique opérationnelle.

4.3.3.2 Les actions

- L'action **skip** est l'instruction vide. Elle est utile pour clarifier le code.
- L'assignation se fait au moyen du symbole **:=** identique à l'assignation dans le langage Pascal.

- Les actions conditionnelles se font au moyen de l'instruction **if ... fi**.

  Soient E l'environnement actuel,
  $cond_1$, ..., $cond_n$ des expressions booléennes,
  $action_1$, ..., $action_n$ une séquence d'actions (avec $n \geq 1$)

  L'exécution de l'instruction :

  if
  $cond_1 \rightarrow action_1$
  ...
  $cond_n \rightarrow action_n$
  fi

  est évaluée de cette manière :

  - $cond_1$ est évaluée en respectant E. Soit v le résultat de l'évaluation.
  - si v = *true*, l'$action_1$ est exécutée en respectant E et l'exécution est terminée ; sinon
  - (a) si $n > 1$, l'instruction

    if
    $cond_2 \rightarrow action_2$
    ...
    $cond_n \rightarrow action_n$
    fi

    est exécutée en respectant E.

    (b) sinon (v = *false* et $n = 1$) et l'exécution est terminée.

- Les actions répétitives sont réalisées avec l'instruction **do ... od**. Sous les mêmes conditions que le paragraphe précédent, l'exécution de l'instruction suivante :

  do
  $cond_1 \rightarrow action_1$
  ...
  $cond_n \rightarrow action_n$
  od

  est évaluée de cette manière :

  - les conditions $cond_1$, ..., $cond_n$ sont successivement évaluées en respectant E jusqu'à ce qu'une soit vraie (*true*). Soit $cond_i$ ($1 \leq i \leq n$) cette condition, alors, l'action $action_i$ est exécutée et l'expression

    do
    $cond_1 \rightarrow action_1$
    ...
    $cond_n \rightarrow action_n$
    od

    est de nouveau exécutée.
  - si aucune des conditions $cond_1$, ..., $cond_n$ n'est évaluée à vraie, l'exécution est terminée.

- Les actions composées sont réalisées avec l'instruction **begin ... end** .

  Soient E l'environnement actuel,
  une séquence d'action $action_1$, ..., $action_n$ ($n > 0$)

  L'exécution de l'instruction :

  begin $action_1$; ... ; $action_n$ **end**

  en respectant l'environnement E consiste à exécuter chacune des actions $action_1$, ..., $action_n$.

4.3.3.3 Le déclenchement de règles

Soit E, l'environnement actuel défini par ses composantes (voir 4.3.3.).
Supposons une règle *rule_name* à n paramètres ($n \geq 0$).
L'exécution de l'action :

trigger off *triggering_mode rule_name*($expr_1$, ..., $expr_n$)

se passe comme suit :

- les expressions $expr_1$, ..., $expr_n$ sont évaluées en respectant E. Soit $L = \{v_1, ..., v_n\}$ la liste de leurs valeurs respectives.
- (a) si le mode de déclenchement (*triggering_mode*) est **for_current**, l'effet de l'exécution est d'ajouter la nouvelle règle *rule_name* à l'ensemble des règles actives DStrig. Donc,

  $DStrig = DStrig \cup \{(R,L)\}$

  où R est la règle déclenchée identifiée par son nom et sa liste de paramètres courants L.

  (b) si le mode de déclenchement est **for_next**, l'effet de l'exécution est de définir un nouvel environnement actuel F identique à E excepté pour DSnext (ensemble des règles à activer pour l'enregistrement suivant) qui devient :

  $DSnext = DSnext \cup \{(R,L)\}$

  où R est la règle déclenchée identifiée par son nom et sa liste de paramètres courants L.

  (c) si le mode de déclenchement est **at_completion**, l'effet de l'exécution est de définir un nouvel environnement actuel F identique à E excepté pour DScompl (ensemble des règles à activer à la terminaison de l'analyse du fichier) qui devient :

  $DScompl = DScompl \cup \{(R,L)\}$

  où R est la règle déclenchée identifiée par son nom et sa liste de paramètres courants L.

  Un exemple sera développé dans la section 4.3.7. pour mieux expliquer concrètement le déclenchement de règles.

4.3.3.4 Les appels de procédures externes

Soit E l'environnement actuel. L'exécution de l'appel de procédure ou de fonction :

procedure_name ($expr_1$, ..., $expr_n$) ($n \geq 0$)

se passe comme suit :

1. les expressions $expr_1$, ..., $expr_n$ sont évaluées en respectant E. Soient v_1, ..., v_n leurs valeurs respectives ;
2. l'appel de procédure :

   procedure_name(v_1, ..., v_n)

   est exécuté en respectant E.

4.3.4 Les variables

Dans le langage RUSSEL, il y a trois types de variables.

4.3.4.1 Les variables locales

Ces variables sont locales relativement à une règle. Elle est déclarée juste après le nom de la règle et la liste de paramètres de cette règle.
L'étendue (*scope*) de ces variables est la déclaration de la règle dans laquelle la variable est déclarée. Le temps de vie de ces variables correspond à l'exécution de la règle.

4.3.4.2 Les variables globales internes

Déclarée au début d'un module, une telle variable a comme portée le module contenant la déclaration de la variable. Le temps de vie de cette variable est l'exécution complète du programme d'analyse.

4.3.4.3 Les variables globales externes

Déclarée au début d'un module, une telle variable a comme portée tous les modules qui composent le programme d'analyse. Le temps de vie de ces variables est l'exécution complète du programme d'analyse.

Dans le chapitre 8, nous développerons la gestion de mémoire du langage RUSSEL car il y existe une erreur, et nous y proposerons des modifications possibles à l'implémentation pour corriger cette dernière.

4.3.5 La librairie de procédures pré-programmées

Il existe une librairie de procédures prêtes à l'emploi disponible à tout utilisateur. Celui-ci n'en voit que la description des paramètres car il n'a pas à s'inquiéter de la manière dont elles ont été implémentées.
Voici la liste des routines C fournies avec le logiciel ASAX.

- createNADF : création d'un fichier NADF ouvert en mode écrit avec le nom donné comme paramètre,
- closeNADF : fermeture d'un fichier NADF,
- writeNADF : écriture d'un enregistrement actuel NADF dans le fichier donné en entrée,
- strToInt : conversion d'une chaîne de bytes en entier,
- println : affichage d'une liste d'arguments sur l'output standard, liste suivie d'un caractère *new line*,
- print : affichage d'une liste d'arguments sur l'output standard (sans le caractère *new line*),
- bytesToInt : conversion d'une chaîne de bytes en la valeur entière décimale qu'ils représentent,
- display_current : imprime sur écran l'enregistrement NADF actuel,
- show_trail : affichage du contenu d'un fichier NADF spécifié.

4.3.6 Conventions de notation

Nous allons utiliser ces notations pour représenter les règles RUSSEL :

- les mots-clés tirés de la grammaire BNF de RUSSEL sont en gras : **if** ... **fi**
- les champs de l'enregistrement courant sont en italique : *CS*
- les paramètres de la règle courante sont en Courier : `filename`

4.3.7 Exemple

Supposons qu'un utilisateur ASAX doit réaliser un programme ASAX réalisant cette spécification : « Si un utilisateur essaie de se logger plus de *occ_par* fois dans le système à partir du même terminal sans succès endéans *within-par* minutes, le message M_1 sera envoyé à l'auditeur. »

Pour réaliser cela, deux règles seront implémentées :

1. La première, *alarm_failed_login* sera activée avec les paramètres appropriés pour envoyer l'alarme requise à l'auditeur quand l'utilisateur ne s'est pas loggé avec succès dans le système *occ_par* fois à partir du même terminal endéans *within_par* minutes.

```
rule alarm_failed_login(occ_par, within_par : integer) ;

if   evt = 'login'
     and res = 'f' →  begin
                          trigger off for_next
                               count_rule (occ_par - 1,
                                           timestp + within_par,
                                           user-id,
                                           station) ;
                          trigger off for_next
                               alarm_failed_login (occ_par, within_par)
                      end ;
     true          → trigger off for_next
                          alarm_failed_login(occ_par,within_par)
fi
```

Expliquons un peu cette règle :

Si l'événement trouvé dans l'enregistrement actuel est un login (evt = 'login') et que le résultat est faux (res = 'f' (false)), alors deux règles sont placées, à cause du **for_next**, dans DSnext (i.e. les règles à activer pour l'enregistrement suivant) : count_rule et alarm_failed_login avec certains paramètres.
Sinon, de toutes manières, la règle alarm_failed_login est activée pour l'enregistrement suivant.
Cette forme de condition avec comme deuxième condition « **true** » arrive souvent. Cela est dû au fait que l'on ne doit jamais oublier d'activer une règle générale (dans ce cas, alarm_failed_login) pour chaque enregistrement du fichier. Si cette condition « **true** » n'était pas là, quand on lance le programme ASAX, si la première condition n'était pas vérifiée, rien ne se passerait et le programme serait déjà terminé car aucune règle n'est activée pour l'enregistrement suivant.

2. La deuxième règle, *count_rule*, activée uniquement par la première règle *alarm_failed_login*, envoie le message M_1 à l'auditeur quand l'utilisateur *user-id-par* n'a pas réussi à se logger dans le système en *l-occ-par* fois à partir du terminal *terminal-par* et que le dernier login infructueux a eu lieu dans un temps < *limit-par*.

```
rule count_rule (l-occ-par : integer ; limit-par, user-id-par,
                                       terminal-par : string) ;

if   l-occ-par = 1
     and evt = 'login'
     and res = 'f'
     and user-id-par = userid
     and station = terminal-par
     and timestp < limit-par
                      → begin
                            send_message(M1) ;
                            v := v + 1              /* counter */
                        end ;
```

evt = 'login'
and *res* = 'f'
and `user-id-par` = *userid*
and *station* = `terminal-par`
and `l-occ-par` > 1
→ **trigger off for_next**
count_rule (`l-occ-par` - 1, `limit-par`, `user-id-par`, `terminal-par`) ;
timestp ≥ `limit-par`
→ **skip** ;
res = 's'
and *evt* = 'login'
and *station* = `terminal-par`
and `user-id-par` = *userid*
→ **skip** ;
true → **trigger off for_next**
count_rule (`l-occ-par`, `limit-par`, `user-id-par`, `terminal-par`)
fi

Expliquons également celle-ci :

- si l-occ-par (le nombre d'occurrences de mauvais login qui reste encore possibles pour l'utilisateur) = 1 et que le login actuel est mauvais (evt = 'login' et res = 'f') et que c'est l'utilisateur userid qui l'a réalisé sur le terminal terminal-par et que le moment où il a réalisé son mauvais login est en deçà de la limite, alors on envoie le fameux message d'alarme à l'auditeur par la procédure send_message, car la spécification de départ est vérifiée,
- si l'on se trouve sous les mêmes conditions (sauf pour le temps) mais que le nombre de mauvais login encore possibles est supérieur à 1, alors on déclenche pour l'enregistrement suivant la même règle mais avec le nombre de mauvais login encore possibles diminué de 1,
- si le temps auquel a été réalisé l'enregistrement actuel dépasse la limite limit-par, on ne fait rien (voici une application du **skip**) ,
- si le résultat du login est bon, on ne fait également rien. Pour ces deux derniers cas, on arrête la procédure de suivi de mauvais login pour l'utilisateur,
- sinon en général (c'est le **true**), quand aucune des conditions précédentes n'est vérifiée, on déclenche la même règle pour l'enregistrement suivant avec les mêmes paramètres. Cela est fait pour que la personne qui a commencé par un mauvais login (voir alarm_failed_login) soit suivie jusqu'à ce qu'elle ne soit plus considérée en faute (i.e. moins de *occ_par* mauvais login en deçà de *within_par* minutes) ou qu'elle ait réalisé la faute et que l'on ait envoyé le message à l'auditeur.

Il reste enfin à lancer ce programme et comme nous l'avions expliqué dans la sémantique opérationnelle, cela se fait par le biais d'une procédure initiale appelée **init_action**. Dans notre cas, elle pourrait se présenter ainsi.

```
init_action ;
begin
    trigger off for_next alarm_failed_login (3,10) ;
    trigger off at_completion echo()
end .
```

Explication :

- Au lancement du programme ASAX, avant de lire le premier enregistrement, le kernel analyse l'**init_action**, dans notre cas, il sait que pour le premier enregistrement, il devra activer la règle *alarm_failed_login* avec les paramètres occ-par à 3 et within-par à 10. Cela signifie que dans, ce cas-là, l'utilisateur n'aura droit, sur le même terminal, qu'à 3 mauvais login en 10 minutes.
- De plus, à la fin de l'analyse de tout le fichier, une règle *echo* sera lancée (par le déclenchement **at_completion**), on peut supposer par exemple que si on a déclaré une variable globale interne v (voir l'explication de la règle *count_rule*) qui compte le nombre de fois qu'il y a eu abus (entendez par là qu'un utilisateur a essayé de se logger 3 fois en 10 minutes sur le même terminal), on pourrait imprimer dans *echo* cette variable pour avertir l'auditeur du nombre de fois que cela s'est passé sur l'étendue du fichier NADF.

4.4 Extensions d'ASAX

Deux types de composantes peuvent être ajoutées dans une application ASAX : les routines I/O et les routines C définies par l'utilisateur. Pour ajouter une de ces deux composantes, il faut les préparer et les intégrer ensuite dans la nouvelle version d'ASAX.
La figure suivante montre comment on pourrait représenter le kernel d'ASAX et les possibilités d'extension qu'il offre à l'utilisateur et celles qui sont déjà fournies avec ASAX.

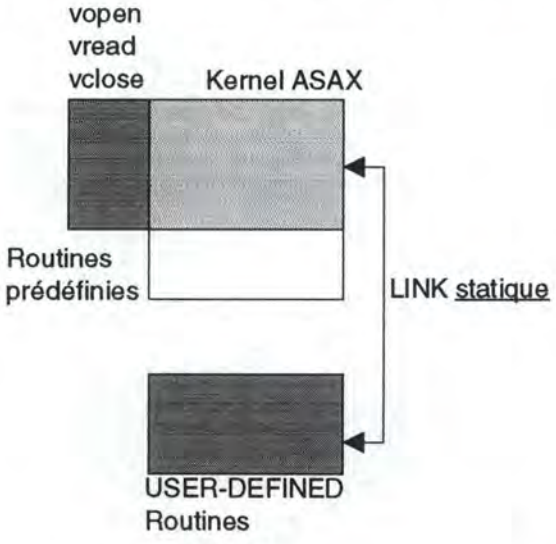


Fig. 4.3: Possibilités d'extension avec ASAX

On peut y voir le kernel ASAX fourni à l'utilisateur avec un ensemble de routines prédéfinies (voir section 4.3.5.). Dans la section 4.2.2., nous avons également présenté la possibilité d'extension pour la génération du fichier audit au moment même de la création des enregistrements, pour une analyse on-line ; cette extension étant réalisée par les trois routines I/O vopen, vread, vclose. Enfin, il est possible à l'utilisateur de personnaliser sa version ASAX

en y ajoutant de nouvelles fonctionnalités, i.e. en créant ses propres routines en C. Ces routines définies par l'utilisateur sont liées au kernel d'ASAX au moment de la compilation, il s'agit donc bien d'un lien statique.

4.4.1 Construction de routines I/O

Ce point a déjà été développé dans la section 4.2.2.

4.4.2 Préparation de routines C

Pour préparer ses propres routines en C, il faut en premier lieu les écrire en langage C en suivant certaines conventions précises. Ensuite, on les compile avec le même compilateur C que celui utilisé pour compiler le kernel ASAX ; on donne leur description dans ce que l'on appelle un fichier de description de librairie C (*C-library description file*) et on génère enfin la nouvelle version d'ASAX.

4.4.2.1 Conventions de paramètres

Pour être intégrés dans une nouvelle version d'ASAX et être manipulés proprement, les arguments des nouvelles routines C doivent suivre un format bien défini. Réciproquement, une liste d'arguments fournies par l'analyseur d'ASAX doit être correctement interprétée par la routine C.
Comme une routine C peut avoir un nombre variable d'arguments, plutôt que de fournir une liste de paramètres, un pointeur vers une zone mémoire contenant la liste de paramètres est fourni.
Il faut donc savoir comment les types de données d'ASAX (entier et string de bytes) sont concrètement représentés et comment ces données sont organisées dans la zone de paramètres de manière à ce qu'elles puissent correctement être analysées par une routine C.

4.4.2.2 Représentation des types de données

Le type entier d'ASAX est simplement représenté comme un entier sur 4 bytes en C. Cette hypothèse sur la taille de l'entier est nécessaire pour éviter des problèmes quand on porte ASAX sur une architecture avec des caractéristiques de taille différentes. Il ne faut pas oublier qu'une des caractéristiques d'ASAX est sa portabilité.

Un string 's' de longueur 'long' est représenté par un pointeur vers une zone mémoire de deux champs. Le premier, sur 2 bytes, est la longueur 'long' de 's' et le second est le string en lui-même.

4.4.2.3 Format d'une zone de paramètres

Une routine C définie par l'utilisateur '$p(p_1,...,p_n)$' est implémentée par une routine C qui reçoit un argument simple qui est l'adresse de la zone mémoire représentant la liste de paramètres. La taille de cette zone est toujours un multiple de 4 bytes pour des raisons de compatibilité. Son organisation dépend du type des paramètres et de leur mode de passage, i.e. par valeur ou par référence (par adresse).

Soit un paramètre donné 'p_i' :

- Si 'p_i' est un paramètre passé par valeur :
  - de type entier, 'p_i' est cet entier sur 4 bytes ;
  - de type string, 'p_i' est le pointeur vers la zone du string ;
- si 'p_i' est un paramètre passé par référence (par adresse) :
  - de type entier, 'p_i' est un pointeur de 4 bytes vers un entier sur 4 bytes ;
  - de type string, 'p_i' est un pointeur de 4 bytes vers une zone de 4 bytes.

La figure suivante exprime les quatre cas précédents.

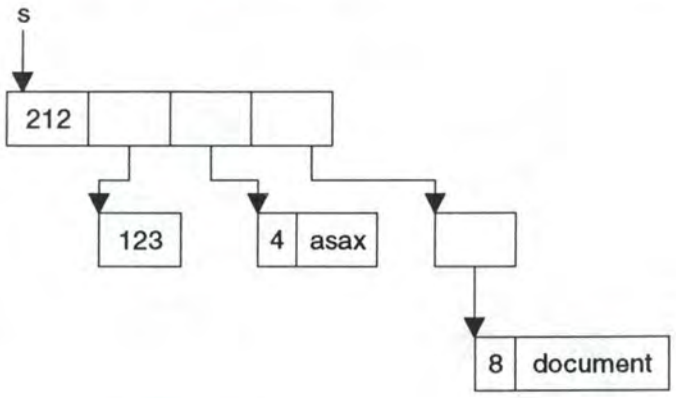


Fig. 4.4 : Zone de paramètres

212	est un paramètre passé par valeur de type entier,
123	est un paramètre passé par adresse de type entier,
asax	est un paramètre passé par valeur de type string,
document	est un paramètre passé par adresse de type string.

4.4.2.4 Implémentation d'une routine C prédéfinie

Soit '$p(v_1, ..., v_n ; r_1, ..., r_n)$' une routine C définie par l'utilisateur avec '$v_1, ..., v_n$' la liste des paramètres passés par valeur et '$r_1, ..., r_n$' la liste des paramètres passés par adresse.
'p' est implémentée par une routine C du même nom :

```
p(s)

char *s.
{
...
}
```

où 's' est l'adresse de la zone de paramètres. La zone est explorée à partir de cette adresse. Comme l'arité de la fonction ou procédure implémentée et le type et le mode de passage de chaque paramètre sont connus grâce au fichier de description de librairie (voir section 4.4.2.5.), il est possible d'aller chercher n'importe quel paramètre en partant de l'adresse 's'.
Si la routine n'a pas de paramètre, l'adresse de la zone de paramètres doit également être fournie, la différence est simplement que la zone ne sera pas scannée par la routine.

4.4.2.5 Le fichier de description de librairie C

4.4.2.5.1 But

Un fichier de description de librairie C (*C-library description file*) a pour but de rassembler une description de haut niveau de toutes les routines C définies par l'utilisateur disponibles pour une version d'ASAX donnée.
Pour ajouter, supprimer une routine définie par l'utilisateur, il suffit de modifier le fichier de description de librairie C et de reconstruire une nouvelle version d'ASAX.

4.4.2.5.2 Syntaxe

Tout fichier de description de librairie C est un fichier texte ayant la syntaxe suivante :

<C-library description file> ::= <empty> |<descriptor list>
<descriptor list> ::= <descriptor> ; ... ; <descriptor>.
<descriptor> ::= <returned type> <func_proc_name> **(** <parameter type list> **) :** <filename>
<returned type> ::= <empty> | **integer** | **string**
<func_proc_name> ::= identifier
<parameter type list> ::= <empty> | <variable_arity> | <param_list>
<variable_arity> ::= **undef :** <passing mode>
<param_list> ::= <parameter type> **,** ... **,** <parameter type>
<parameter type> ::= <type> **:** <passing mode>
<type> ::= **integer** | **string**
<passing mode> ::= **val** | **ref**
<filename> ::= identifier

4.4.2.5.3 Sémantique

Dans un fichier de description de librairie, une routine C est spécifiée par ses caractéristiques :

- le type de la valeur retournée (s'il y en a une),
- le nom de la routine,
- l'arité (i.e. le nombre de paramètres),
- le type et le mode de passage de chaque argument. Il est à remarquer qu'**undef** est utilisé pour exprimer le fait que soit le nombre de paramètres est inconnu, soit le type d'un ou plusieurs paramètres est inconnu,
- le nom du fichier du module objet contenant la routine compilée.

Evidemment, deux routines prédéfinies n'ont jamais le même nom et tous les noms de fichiers doivent correspondre à des noms de fichiers existants.

4.4.2.5.4 Exemple

Voici le fichier de description de librairie que nous avons créé pour l'extension d'ASAX dans le cadre de notre stage à Hambourg, extension qui s'occupe de la gestion de tables associatives (voir chapitre 5).

```
/* Associative table management */

integer         create_TA(undef:val)          : tassoc;
integer         isMember(undef:val)           : tassoc;
integer         install_1(undef:val)          : tassoc;
integer         install_g(undef:val)          : tassoc;
integer         update(undef:val)             : tassoc;
integer         retrieve_1(undef:ref)         : tassoc;
integer         retrieve_g(undef:ref)         : tassoc;
integer         Remove(undef:val)             : tassoc;
integer         release(integer:val)          : tassoc;
integer         print_TA(integer:val)         : tassoc.
```

4.5 L'avenir d'ASAX

ASAX, par son universalité (il ne dépend d'aucune plate-forme particulière et utilise des adaptateurs de format pour traduire les flux de données dans son propre format) a de l'avenir devant lui. Durant notre stage, nous l'avons utilisé abondamment pour analyser les flux de données en provenance de l'émulation d'un virus sous DOS. Comme nous le verrons plus loin, ASAX nous a permis d'automatiser la classification des virus, de plus en plus nombreux à travers le monde. Nous allons principalement parler d'une analyse off-line, i.e. une analyse qui ne se fait pas en temps réel, mais par après, quand l'exécution de ce que l'on veut analyser a déjà eu lieu.
Mais ASAX ne fait pas que de l'analyse off-line, il existe tout le côté on-line déjà développé (voir section 4.2.2.) qui permet de faire de l'analyse en temps réel et permet ainsi de réagir plus rapidement aux abus et coups malicieux. Car il ne faut pas oublier que dans le cas d'une analyse off-line, on ne sait plus rien faire si le système a été attaqué, on ne peut qu'en chercher les raisons et réagir pour que cela ne se reproduise plus.

Mais ASAX n'en reste pas là, une nouvelle version vient de sortir pour les systèmes distribués. Il fallait bien y arriver un jour car les réseaux sont de plus en plus complexes et leur sécurité également. Deux faits ont poussé à passer à la version distribuée :

- La corrélation d'actions d'utilisateurs ayant lieu sur des machines différentes pourrait révéler un comportement malicieux tandis que les mêmes actions pourraient sembler légitimes si elles étaient considérées au niveau d'une machine seule. Si on reprend l'exemple développé à la section 4.3.7., on pourrait vérifier si l'utilisateur n'essaie pas de se logger à partir de machines différentes.
- Le monitoring de sécurité de réseau peut fournir potentiellement un renforcement plus cohérent et plus flexible d'une politique de sécurité donnée. Par exemple, la personne qui s'occupe de la sécurité peut établir une politique de sécurité commune pour toutes les machines auditées mais choisir de renforcer les mesures de sécurité pour des machines critiques ou pour des utilisateurs suspects.

Le système distribué utilise RUSSEL pour filtrer les données d'audit à chaque machine auditée et analyser les données filtrées rassemblées sur une machine centrale. Le langage étant le même aux différents niveaux (local et central), cela nous fournit un produit pour un contrôle flexible et de granularité graduelle à différents niveaux : utilisateur, hôtes (machines), sous-réseaux, domaines, etc.
Comme on peut le voir, ASAX a encore de l'avenir devant lui.

5. Gestion de tables associatives en langage RUSSEL

Comme expliqué auparavant dans le chapitre sur ASAX, si l'on veut garder des renseignements sur des enregistrements passés, il faut passer ces renseignements en paramètres dans les règles pour ne pas en perdre la trace. Mais à la longue, et c'est un gros problème, la liste des paramètres que l'on veut garder, en général, ne cesse d'augmenter plus on avance dans le fichier audit à analyser.
Supposons que l'on recherche la présence d'un virus compagnon dans un fichier audit. Lors de la création d'un fichier COM, il faudra que l'on ait gardé en paramètre tous les fichiers EXE qui auront été analysé pour voir si ce fichier EXE existe déjà. On peut comprendre que cette liste peut vite grandir.
C'est pourquoi, avant de partir à Hambourg, nous avons implémenté en C des procédures prédéfinies pour la gestion de tables associatives en langage RUSSEL.

5.1 Définitions préliminaires

1. I désigne l'ensemble des valeurs entières en RUSSEL. S désigne l'ensemble des valeurs string en RUSSEL ;
2. Une table associative T est une partie du produit cartésien $(M_i)_{1\le i\le n} = M_1 \times ... \times M_n$ où M_j = I ou S. Par extension, on identifie $(M_i)_{i=1}$ avec M_1.
   Exemples : $T_1 = \{1,2\}$; $T_2 = \{$'virus detected','Vienna virus'$\}$
3. Par restriction, on se limitera aux tables associatives où la première projection est une clef. A chaque table associative $T \subseteq (M_i)_{1\le i\le n}$ est associé un ensemble de projections $(p_i)_{1\le i\le n}$ de la façon habituelle :
   $p_j : T \subseteq (M_i)_{1\le i\le n} \rightarrow M_j$
   $e = (e_1,...,e_j,...,e_n) \rightarrow e_j = p_j(e)$
   On a donc $e \in T \wedge f \in T \wedge p_1(e) = p_1(f) \Rightarrow e = f$.
   Dans la suite, la projection p_1 relative à une table associative T est notée *key* de sorte que key(e) désigne la clef d'accès de l'élément e de T.
4. Réciproquement, étant données une table T et une clef k (de type entier ou string), on s'intéresse également à l'élément e de T lorsqu'il existe, tel que key(e) = k.
   On définit alors la fonction n (réciproque de key) :
   n : I ou S $\rightarrow$ $T = (M_i)_{i\le n}$
   k $\rightarrow n(k) = e$ t.q. key(e) = k
5. A toute table T est associé de manière univoque un numéro (de type entier) qui l'identifie par rapport à toutes les tables. Il est noté Id(T).
6. Une clef est identifiante lorsque l'on se restreint à une table donnée. Autrement dit, deux éléments appartenant à deux tables différentes peuvent avoir la même clef.
7. Il en résulte que, étant donné un ensemble de tables associatives, un élément e est entièrement défini par le couple (Id(T), key(e)) où T est la table identifiée par Id(T) et contenant l'élément e.

5.2 Description de l'implémentation

5.2.1 Implémentation d'une table associative

Une table associative est implémentée par une table de hash pour accélérer l'accès à un élément donné. La valeur de hash est calculée sur la clef d'accès. La taille de la table de hash doit obligatoirement être un nombre premier, soit HSIZE.

- La valeur de hash d'un string s = '$b_1b_2...b_n$' est définie par
  $hash(s) = (hashval) \bmod (HSIZE)$.
  hashval est une fonction récursive définie par : $hashval = b_i + 31 * hashval$, pour i variant de de 1 à n.
- La valeur de hash d'un entier i est définie par $hash(i) = |i| \bmod (HSIZE)$

De cette manière, pour accéder à l'élément e de clef k = key(e), on calcule sa valeur de hash. Cette valeur (comprise donc entre 0 et HSIZE - 1) nous sert d'index dans la table de hash. Une entrée de la table de hash est une liste de tous les éléments ayant la même valeur de hash. Pour accéder à un élément particulier, on parcourt la liste.

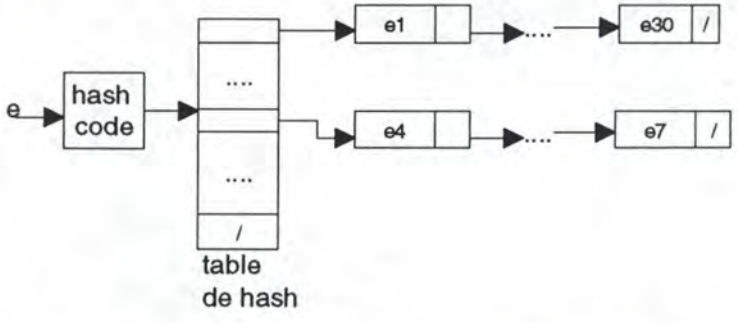


Fig. 5.5 : Représentation d'un ensemble par hash code

La création d'une table associative vide revient à allouer dynamiquement une table de hash dont les entrées contiennent toutes la valeur pointeur nul.

5.2.2 Implémentation de l'ensemble des tables associatives

Une table statique *sets* est utilisée pour implémenter l'ensemble de toutes les tables associatives. Chaque entrée de cette table contient les informations relatives à une table associative donnée. Cette entrée contient entre autres, l'adresse de la table de hash qui représente cette table associative et une représentation du type des éléments de cette table.
La représentation du type est implémentée par un entier sur 32 bits. Lorsqu'un bit est à 0, le type est entier sinon il est string.
Supposons que $e = (e_1, ..., e_n)$ avec $n \leq 32$, alors les n premiers bits des 32 bits en commençant par la droite indique le type, les autres étant mis à 0. Par exemple, si $T \subseteq S \times I$, alors le type de T est représenté par l'entier sur 4 bytes 00 00 00 02.
Il en résulte que l'arité d'une table associative est limitée à 32.
Id(T) est l'indice de T dans *sets*.

Lorsqu'une table associative est créée, une entrée de la table statique *sets* est initialisée indiquant l'adresse de la table de hash correspondante, l'entier représentant le type et son arité.

Les entrées non utilisées de *sets* sont chaînées pour optimiser l'accès à une entrée « libre ». A chaque instant, on connaît l'indice dans *sets* de la tête de la liste chaînée d'entrées non utilisées (*freeListHead*).

5.2.3 Opérations sur les tables associatives

A la fin de chaque opération se trouve entre parenthèses le nom de la routine C.

5.2.3.1 Création d'une table associative

La création d'une table vide revient à allouer dynamiquement sa table de hash et à rajouter une entrée à la table statique *sets* pour stocker son type (1 pour les strings, 0 pour les entiers) et son arité. S'il reste assez de place sur cette dernière table, tout se passe bien, sinon, un message d'erreur est affiché (**create_TA**).

5.2.3.2 Ajout d'un élément

Il y a deux types d'ajouts possibles.
Etant donné une table T, si l'on a toutes les valeurs e_1, ..., e_n d'un élément e, l'ajout de cet élément e de clef k = e_1 revient à rajouter l'élément e = (e_1, ..., e_n) à T (**install_g**).
Si, par contre, l'on n'a que la clef k, l'ajout d'un élément ayant cette clef revient à rajouter l'élément e = (k, 0, ...,0) à T (**install_1**).
Si un élément de clef k existe déjà dans la table, un message d'erreur est affiché.

5.2.3.3 Suppression d'un élément

Soit une table T d'identificateur Id(T), la suppression d'un élément e de clef k revient à désallouer la mémoire dynamique représentant l'élément e dans la table de hash associée à T. Si T ne contient pas d'élément de clef k, un message d'erreur est affiché (**Remove**).

5.2.3.4 Mise à jour d'un élément

Cette opération permet de modifier la valeur d'une projection donnée d'un élément donné par sa clef appartenant à une table d'un identifiant donné (**update**).

5.2.3.5 Lecture de la valeur d'un élément

Ici, on accède à la valeur d'un élément donné par sa clef et appartenant à une table ayant un identificateur donné. On distingue deux opérations de lecture :

1. une opération permettant de lire la $i^{ème}$ projection e_i d'une élément e = (e_1, ..., e_i, ..., e_n) (**retrieve_1**) ;
2. une opération de lecture globale permettant de lire toutes les valeurs e_1, ..., e_n (**retrieve_g**).

5.2.3.6 Appartenance

Cette opération permet de tester si un élément de clef k appartient à une table T d'identificateur Id(T) (**isMember**).

5.2.3.7 Destruction d'une table

L'effet de cette opération est de désallouer toutes les zones dynamiques relatives aux éléments de la table, de désallouer la table de hash elle-même et enfin, de rajouter son entrée dans la table *sets* à la liste chaînée d'entrées non encore utilisées (**release**).

5.3 Exemple

Reprenons l'exemple du chapitre 4. Rappelons la spécification du programme à réaliser : « Si un utilisateur essaie de se logger plus de *occ_par* fois dans le système (à partir du même terminal) sans succès endéans *within_par* minutes, le message M_1 sera envoyé à l'auditeur. »

Un des problèmes possibles de la solution du chapitre précédent est que si un utilisateur se logge mal, puis qu'un autre utilisateur utilise le même terminal et se logge mal et qu'enfin, un troisième utilisateur arrive (cela peut être un des deux premiers), il y a un risque d'explosion du nombre de règles tournant en même temps pour l'analyse d'un enregistrement courant. Une règle s'occupe de vérifier si on est dans le cas du premier utilisateur (si on est dans la zone des *within_par* minutes, la règle count_rule tourne toujours), une autre s'occupe du deuxième utilisateur et une troisième vérifie si ce n'est pas un nouvel utilisateur qui arrive.
La table associative est une solution à ce risque car on ne doit pas faire tourner un grand nombre de règles par enregistrement.
Voici le corps des nouvelles procédures possibles avec l'ajout des tables associatives avec les mêmes conventions de notation que dans la section 4.3.7.

```
rule alarm_failed_login(occ_par, within_par : integer) ;
var user : string ; limit, occ_left : integer ;
begin
if      evt = 'login'
        and res = 'f'
        → begin
                c1 := isMember(t, user-id) ;
                if
                        c1 = 0
                        → begin
                                user := user-id ;
                                c2 := retrieve_g(t, user, limit, occ_left) ;
                                if
                                        occ_left = 1 and limit > timestp
                                        → send_message(M1) ;
                                        occ_left = 1
                                        → begin
                                                c3 := Remove(t, user-id) ;
                                                c3 := install_g(t, user-id, timestp+within_par,
                                                                occ_par - 1)
                                           end ;
                                        limit < timestp
                                        → begin
                                                c3 := Remove(t, user-id) ;
                                                c3 := install_g(t, user-id, timestp+within_par,
                                                                occ_par - 1)
                                           end ;
                                        true
                                        → c3 := update(t, user-id, 3, occ_left -1)
                                fi ;
                        true
                        → c2 := install_g(t, user-id, timestp + within_par,
                                        occ_par -1)
           end ;
        evt = 'login'
        and res ='s'
                → begin
```

```
                    c1 := isMember(t, user-id) ;
                    if c1 = 0
                    → c2 := Remove(t, user-id)
                    fi
                end
fi ;
trigger off for_next alarm_failed_login (occ_par, within_par)
end
```

Explication :

- Si l'événement trouvé dans l'enregistrement actuel est un login (evt = 'login') et que le résultat est mauvais (res = 'f')
    - si l'utilisateur fait déjà partie de la table associative t, on recherche ses composantes (retrieve_g) ;
        - si l'utilisateur fait son mauvais login en deçà de la limite et qu'il a dépassé le nombre de mauvais login acceptés, on envoie le message d'alarme à l'auditeur par la procédure send_message ;
        - si l'utilisateur fait un mauvais login pour la dernière fois possible mais qu'il est hors de la limite de temps, on le réinstalle avec une nouvelle limite de temps ;
        - si l'utilisateur est au delà de la limite qui lui est accordée, on le réinstalle également avec une nouvelle limite de temps ;
        - sinon, on lui retire une des chances qu'il a encore ;
    - sinon, on l'installe dans la table associative en tant que personnage malveillant potentiel ;
- par contre, si le login est bon, on vérifie qu'il ne fait pas déjà partie de la table associative, et si c'est le cas, on l'en retire vu qu'il n'est plus en faute.

Ensuite, on place alarm_failed_login dans DSnext pour que cette règle soit activée pour l'enregistrement suivant.

```
global t : integer ;
init_action ;
begin
        t := create_TA (1,0,0) ;                /*(user, limit, occ-left)*/
        trigger off for_next alarm_failed_login(3,10) ;
        trigger off at_completion release(t) ;
        trigger off at_completion echo()
end.
```

Explication :

- Comme expliqué dans le chapitre précédent, le kernel analyse l'**init_action**. La première action à effectuer est la création d'une première table associative composée de :
    1. l'utilisateur (clef identifiante) ;
    2. le temps limite pour cet utilisateur ;
    3. le nombre de bad login encore possibles.

    On ne s'occupe plus du terminal sur lequel l'utilisateur se logge car, pour le cas du chapitre 4, il suffit que l'utilisateur change de terminal après deux essais infructueux et il peut recommencer sa tentative de fraude ailleurs.

On aurait pu faire une tout autre table associative, par exemple, sur le terminal où, dans ce cas, on rechercherait le nombre de mauvais login sur le même terminal endéans un certains temps quel que soit la personne qui tente de se logger. Il se pourrait en effet qu'une personne malveillante essaie de pénétrer le système en utilisant le nom de tous les copains de sa classe sans connaître leur mot de passe. Pour ce cas-là, la création de la table associative aurait été : create_TA (1,0,0) /*(term, timestp, occ-left)*/.

L'utilisateur a disparu de la table vu que l'on n'en tient plus compte.

- Il faut évidemment libérer toute la zone mémoire allouée dynamiquement au moment de l'utilisation de la table associative. Cela se fait par la routine **release**(c1) (où c1 est l'identifiant Id(T) de la table associative T) à la fin de l'analyse du fichier audit (at_completion).
- Le reste est le même que dans le cas précédent.

6. Analyse dynamique de virus informatiques

6.1 Introduction

Après avoir expliqué de manière théorique les deux systèmes (ASAX et PANDORA) que nous avons utilisé durant notre stage à Hambourg, nous allons maintenant passer à la présentation de notre travail réalisé durant le stage.

Dans ce chapitre, nous présenterons l'analyse statique par le biais de deux approches pour la détection des virus. Ensuite, nous développerons plus amplement le mécanisme de la détection dynamique, en présentant deux méthodes de représentations des règles de détection utilisées pour le concept VIDES, et en expliquant les différentes règles que nous avons créées durant le stage.

Ce chapitre présentera donc la modélisation de stratégies d'infection à l'aide de diagrammes. Le chapitre 7 montrera comment les règles RUSSEL peuvent en être dérivées.

6.2 Analyse statique

Chaque mois, les chercheurs doivent se débrouiller avec des milliers de fichiers suspects. Le problème n'est pas tellement le nombre de nouveaux virus, nombre qui n'augmente que de quelques centaines par mois mais, c'est le nombre de fichiers que le chercheur reçoit et qu'il doit traiter qui cause les problèmes. Parmi une centaine de fichiers, un seul peut-être contient réellement un nouveau virus. Malheureusement, il n'y a pas d'autres possibilités que d'analyser tous les fichiers.

Aujourd'hui, l'analyse statique peut être divisée en deux approches : l'approche spécifique au virus (*virus specific*) et l'approche générique (*generic*). En principe, la première approche requiert une connaissance des virus avant qu'ils ne soient détectés. Ce type de technologie est connu de tous sous le nom de *scanner* (voir section 2.5.1.1.). La deuxième approche tente de détecter un virus par l'observation des attributs caractéristiques à tous les virus. Par exemple, les vérificateurs d'intégrité (*integrity checkers*) détectent les virus par la vérification de modifications dans les fichiers exécutables.

6.2.1 Détection spécifique de virus

La détection spécifique de virus est de loin le type le plus populaire de protection contre les virus. L'information tirée de l'analyse du virus est utilisée dans le scanner pour détecter le virus. Généralement, un scanner utilise une base de données contenant des informations d'identification de virus de telle manière qu'il puisse détecter tous les virus qui ont été analysés précédemment. Le terme *scanner* n'est pas tout à fait juste, il faut parler de scanner lexical, i.e. un outil de vérification d'échantillons. Traditionnellement, le scanner était seulement cela. L'information extraite des virus était des chaînes représentatives de ce virus particulier. Cela signifie que la chaîne doit être :

- différente significativement de toutes les autre chaînes caractéristiques des autres virus, et
- différente significativement des chaînes trouvées dans les programmes non malicieux.

Découvrir de telles chaînes était l'art de l'écriture d'anti-virus jusqu'à ce que les virus chiffrés apparaissent. Ceux-ci furent le premier défi aux méthodes de recherche de chaînes. Le corps du virus était chiffré, et ne pouvait être cherché à cause de sa nature variable. Cependant, le corps était précédé d'un décrypteur qui devait être en texte *clair* (non chiffré) sinon il ne pouvait être exécuté. Ce décrypteur pouvait toujours être détecté en utilisant les méthodes de recherche de chaînes même si cela devenait difficile de différencier des virus différents.

Les virus polymorphiques représentent l'étape suivante dans l'évitement de la détection. Ici, le décrypteur est implémenté de manière variable de telle sorte que la recherche d'échantillons est devenue impossible ou très difficile. Les premiers virus polymorphiques étaient identifiés par un ensemble de chaînes à éléments variables.

De plus, les techniques de détection de virus simple sont rendues non fiables par l'apparition de moteurs de mutation (*mutation engines*) tels MtE et TPE (*Trident Polymorphic Engine*). Ceuxi-ci sont des modules de librairie d'objets générant des implémentations variables du décypteur du virus. Ils peuvent être facilement liés avec des virus pour produire des infecteurs hautement polymorphiques. Les techniques de scanning sont donc devenues compliquées par le fait que les virus résultant n'ont aucune chaîne en commun même si leur structure sous-jacente reste constante.

Il y a quelques années, les meilleurs des scanners sont passés du stade de la détection de virus à la tentative d'identification de virus. Cela est souvent réalisé à l'aide de chaînes additionnelles, dépendantes parfois de la position, ou de checksums sur des portions invariables de virus. En plus de cela, beaucoup d'anti-virus ont implémenté des émulateurs de code machine pour que le décrypteur propre au virus puisse être utilisé pour le décrypter. En utilisant ces petites améliorations, l'identification positive de virus - même polymorphiques - ne pose pas de problème.

L'étape suivante que beaucoup de scanners ont suivie est le passage de la détection de virus connus à la détection de virus inconnus. La méthode de choix est heuristique. Les heuristiques construites dans l'anti-virus, essaient d'inférer si un fichier est infecté ou non. Cela est réalisé le plus souvent en recherchant des échantillons de certains fragments de code qui apparaissent souvent dans les virus et pas dans les programmes non malicieux. Seulement, les heuristiques souffrent d'un taux modéré à élevé de fausses alarmes (*false positives*), i.e. la reconnaissance fausse de l'existence d'un virus. Bien sûr, un fournisseur de scanner heuristique améliorera ses heuristiques de manière à éviter les fausses alarmes et toujours trouver tous les nouveaux virus, mais ces deux buts ne pourront jamais être entièrement réalisés.

6.2.2 Détection générique de virus

Par définition, pour être des virus, les virus informatiques doivent se reproduire. Cela signifie donc qu'un virus doit pouvoir être observable par son mécanisme de reproduction.
Malheureusement, l'observation de la reproduction n'est pas aussi facile qu'elle semble l'être. En effet, DOS, dans ses différentes versions, ne fournit pas d'isolation de processus ou même de protection du système d'exploitation vis-à-vis des programmes utilisateurs. Cela signifie que tout programme de monitoring peut être contourné par un virus qui a été programmé pour le faire. Beaucoup d'anti-virus essayaient de suivre l'activité du système pour y trouver des virus, mais ils n'étaient pas garantis contre tous les virus. Ce problème mena à la mort de nombreux outils de ce type.

Une approche plus fréquente est la détection de symptômes de l'infection comme par exemple, les modifications de fichiers. Ce type de programme est généralement appelé un vérificateur

d'intégrité (*integrity checker* ou *check-summer*). Quand des programmes sont installés sur un PC, des checksums sont calculés sur le fichier entier ou parfois sur des portions de celui-ci. Ces checksums sont alors utilisés pour vérifier que les programmes n'ont pas été modifiés. Le défaut de cette méthode est que le vérificateur d'intégrité peut détecter une modification dans le fichier, mais ne peut déterminer si la modification est due à un virus ou non. Une modification légitime à la zone des données du programme par exemple, causera la même alarme que l'infection par un virus.

Un autre problème est la technologie des virus construits spécifiquement contre des anti-virus. Les progrès dans les techniques stealth et tunneling rendent les mises à jour nécessaires. Il y a également eu des attaques directes contre des vérificateurs d'intégrité particuliers (voir section 2.5.1.3.), les rendant inutiles. De nouveau, le manque de support du système d'exploitation rend la prévention de ce type d'attaque très difficile. En conséquence, l'acceptation de ce type de produit est faible.

La nature non spécifique de la détection a peu d'attrait pour beaucoup d'utilisateurs. Même les outils de réparation générique dans les anti-virus n'aident pas, malgré le fait que ces méthodes rendent effectivement l'identification non nécessaire. Mais le problème est partiellement compréhensible. L'utilisateur est inquiet pour ses données. Simplement désinfecter les programmes ne suffit pas si les données ont été manipulées. C'est seulement si le virus a été identifié et analysé que l'utilisateur détermine si ses données ont été menacées.

Mais, la technique de détection générique de virus ne doit pas être abandonnée pour autant. Elle est aussi valide que la technique de détection spécifique de virus. Les problèmes qui se sont présentés jusqu'à présent proviennent manque de mécanismes de contrôle d'accès et de protection du système d'exploitation sous-jacent, DOS, et des limites des programmes.

6.3 Analyse dynamique

L'analyse dynamique consiste à observer le comportement du virus à l'exécution.

Le concept VIDES utilise trois types de règles de détection pour l'analyse dynamique : les règles de détection génériques (*generic rules*), les règles spécifiques à un virus (*virus specific rules*) et d'autres règles (*other rules*). Les règles génériques sont utilisées pour détecter tous les virus qui utilisent un modèle d'attaque connu. Les règles spécifiques à un virus utilisent de l'information provenant d'une analyse préalable pour détecter ce virus spécifique ou des variantes directes de celui-ci. Ces règles sont similaires aux programmes de détection spécifique de virus si ce n'est qu'ils analysent le comportement dynamique du virus au lieu de son code. Il reste enfin les autres règles qui sont utilisées pour glaner de l'information sur le virus, information qui sera utilisée pour sa classification.

6.3.1 Règles génériques de détection dynamique

Dans le développement d'une règle générique de détection de virus, on a besoin d'un modèle du scénario d'attaque du virus. Un tel modèle n'est pas unique car les virus DOS peuvent choisir parmi beaucoup de stratégies efficaces. Cela s'explique par la diversité des types de fichiers exécutables dans le DOS. Heureusement pour nous, la majorité des virus choisissent une stratégie particulière et infectent seulement un ou deux types de fichiers exécutables.

6.3.1.1 Hypothèses

Deux hypothèses[5] sont faites à propos du comportement des virus DOS, hypothèses qui permettront de construire les règles.

Hypothèse 1 :

> *Un virus fichier modifie le fichier hôte de telle manière que le virus gagne le contrôle du fichier hôte quand celui-ci est exécuté.*

Hypothèse 2 :

> *Le virus dans un fichier infecté reçoit le contrôle du fichier avant que celui-ci ne le reçoive.*

La première hypothèse est tirée de la définition même du virus ; cependant, cette hypothèse ne spécifie pas quand le virus gagne le contrôle sur le fichier hôte.
La deuxième hypothèse signifie que quand le fichier infecté est exécuté, le virus est exécuté en premier lieu et le programme hôte ensuite.

Cette deuxième hypothèse est nécessaire car la première hypothèse ne permet pas de dire pourquoi le virus doit prendre le contrôle avant le programme hôte. Cette hypothèse s'explique par le fait que si le virus n'est pas exécuté en premier, il court le risque de ne jamais être exécuté lors de l'exécution du programme hôte. Au pire, il se peut que l'insertion du corps du virus dans le programme hôte endommage celui-ci à un point tel que le programme hôte ne soit plus exécutable après infection ; cela pourrait arriver par exemple lors de l'infection par un virus réinscripteur (voir section 2.4.1.).

6.3.1.2 Première modélisation : diagramme de transition d'états

Pour représenter les scénarios d'infections par les virus, une méthode est la représentation par les diagrammes de transition d'états. Dans cette représentation, nous distinguons deux composantes :

- un noeud dans un diagramme d'états représente quelques aspects de l'état du système ;
- les arcs représentent les actions effectuées par un programme en exécution.

Il y a également une troisième forme utilisée dans les schémas : le message envoyé à l'utilisateur.

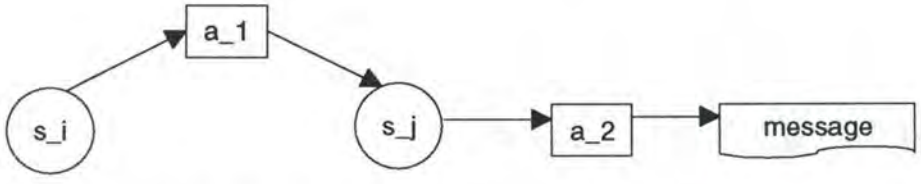


Fig. 6.1 : Diagramme de transition d'états

Soit un état s_i, l'action a_1 a fait passer le système de l'état s_i à l'état s_j comme montré à la Fig. 6.1. Ensuite, une deuxième action a_2 a déclenché l'envoi d'un message à l'utilisateur.

[5] Voir *Dynamic detection and classification of computer viruses using general behaviour patterns*, Baudouin Le Charlier, Abdelaziz Mounji, Morton Swimmer

Le processus d'infection effectué par un virus peut être vu comme une séquence d'actions qui conduit le système d'un état initial *propre* à un état final *infecté* où quelques fichiers sont infectés. De plus, pour obtenir une description complète du scénario, un état est garni d'un ensemble de paramètres caractérisant les objets touchés par les actions. Mais, en pratique, nous ne représentons ces actions que lorsqu'elles sont vraiment pertinentes pour le scénario d'infection.

En résultat, un grand nombre d'actions possibles pourrait avoir lieu entre deux états adjacents, mais ne sont pas prises en compte parce qu'elles n'entraînent pas une modification de l'état actuel du point de vue de l'infection. En termes d'audit, des enregistrements audit non pertinents pourraient être présents dans la séquence d'enregistrements représentant la signature de l'infection.

6.3.1.3 Exemple : infection de fichiers COM

Morton Swimmer avait développé cette règle pour rechercher l'infection de fichiers COM. Il y a présenté deux stratégies possibles :

- le virus est réinscripteur (voir section 2.4.1.). Dès lors, on recherche une écriture en début de fichier (*write BOF* pour *write at beginning of file*) sans lecture préalable au même endroit. Les écritures et lectures à d'autres endroits que le début de fichier (autres que BOF) sont permises.
- Le virus est non-réinscripteur. On s'attend alors à une lecture du début de fichier (*read BOF*) et ensuite une écriture au début de fichier. Avant, entre et après ces deux événements, d'autres types de lecture et écriture sont permis.

L'hypothèse faite ici est que dans les deux cas, l'écriture en début de fichier permet au virus de prendre le contrôle lors de l'exécution.
Dans le cas d'un virus non-réinscripteur, nous présumons que le virus lit premièrement le code original en début de fichier (voir section 2.3.2.1. pour la structure des fichiers COM) et le remplace alors par son propre code ; c'est d'ailleurs généralement un *jump* vers le corps du virus. Dans de nombreux cas, le nombre de bytes lus sera le même que le nombre de bytes écrits, mais ce n'est pas vrai dans le cas général.
Dans le cas d'un virus réinscripteur, le code n'est pas lu et sauvé ailleurs mais écrasé.

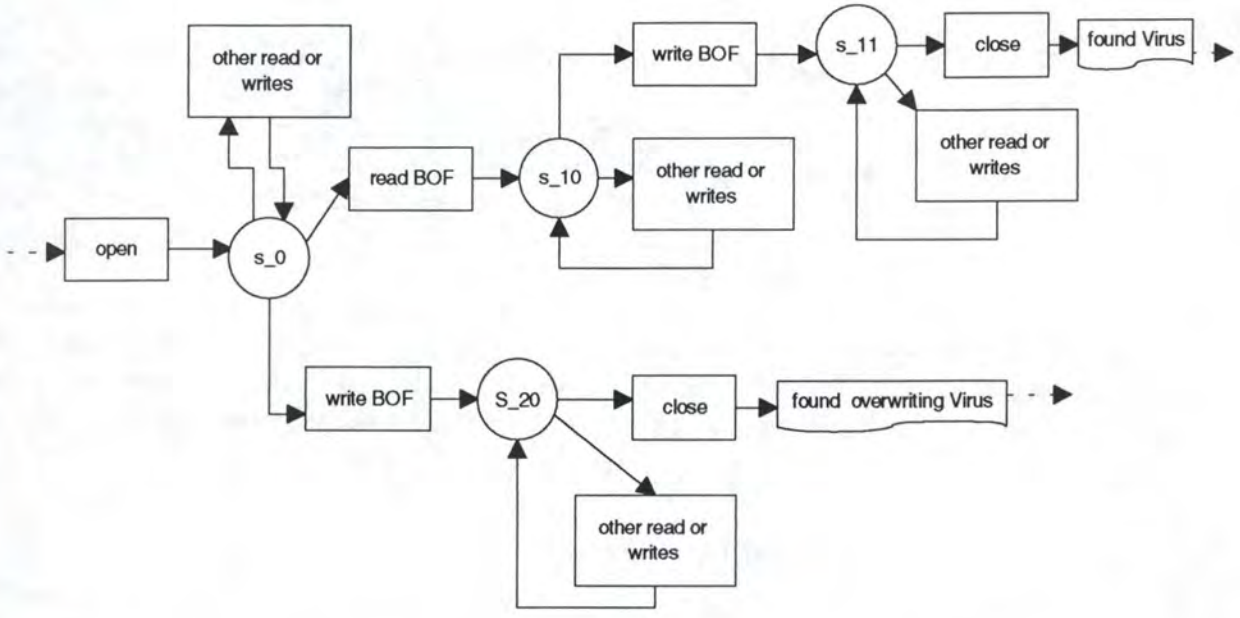


Fig. 6.2 : Infection de fichiers COM : méthode 1

La règle est initalisée par l'ouverture d'un fichier et la terminaison se fait par la fermeture du fichier - fermeture qui n'est pas nécessairement effectuée par le virus. Entre ces deux événements, nous nous attendons à l'infection. Il ne faut pas oublier qu'entre ces deux événements, d'autres événements peuvent avoir lieu. Nous cherchons le *read BOF* suivi du *write BOF* ou le *write BOF* sans la lecture.

6.3.1.4 Deuxième modélisation : diagramme des traitements

Nous allons maintenant présenter une autre approche.

Les schémas contiennent dans cette modélisation trois types de composantes :

- le parallélogramme qui permet de représenter des traitements de recherche de données passées.
- le rectangle représentant les actions : cette forme est utilisée pour représenter la fonction initialisatrice de la règle générique considérée,
- les arcs (avec commentaires) pour passer d'une action à un traitement, ou d'un traitement à un traitement.

Il reste une dernière forme qui est celle du message envoyé à l'utilisateur.

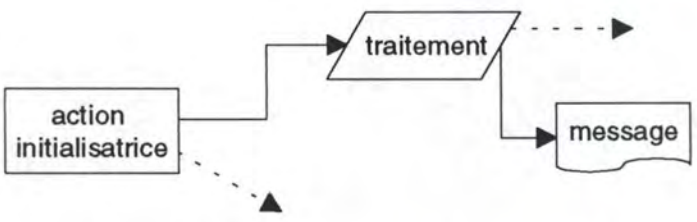


Fig. 6.3 : Diagramme des traitements

La figure nous montre une action initialisatrice déclenchant donc la règle générique. Cette action entraîne un traitement ou autre chose (flèche en pointillés), lui-même déclenchant l'envoi d'un message à l'utilisateur ou autre chose.

Un point commun aux deux approches est leur style de raisonnement. En effet, pour construire les règles génériques, nous devions décider auparavant quel style de virus nous allions essayer de détecter. Si, par exemple, nous décidions de détecter un certain type de virus, nous recherchions les étapes importantes de l'infection pour ce virus. Nous partions donc de notre but final (par exemple, la détection de virus compagnons) et reconstruisions le schéma d'infection du virus. On peut donc dire que ces deux méthodes utilisent un raisonnement *top-down*.

La différence réside dans l'analyse de l'information elle-même. En effet, la première méthode de représentation par les diagrammes de transitions d'états reprend les différentes étapes du raisonnement réalisé auparavant mais réalise une analyse *bottom-up*. En partant de l'événement déclencheur de l'infection (qui est la dernière étape découverte par le raisonnement), il suffit de reconstruire le cheminement étape par étape pour atteindre le but final qui avait été fixé dans le raisonnement. Cette méthode suit vraiment l'ordre chronologique de l'infection des virus, elle présente l'infection du virus.
La seconde méthode de représentation par les diagrammes des traitements réalise quant à elle une analyse *top-down*, i.e. elle reprend le raisonnement réalisé auparavant et le schématise de la même façon en le décomposant en sous-problèmes. La dernière étape du raisonnement (avant d'atteindre le but final) devient l'action initialisatrice de la règle et les autres étapes se retrouvent dans le schéma dans l'ordre chronologique inverse par rapport au schéma d'infection sous la forme de traitements de recherches de données passées. Dans cette méthode, on modélise la détection de l'infection.

6.3.1.5 Elaboration des schémas d'attaque

Dans cette élaboration des schémas de détection d'attaques, nous allons précisément utiliser la deuxième méthode de diagramme des traitements.

6.3.1.5.1 Infection de fichiers exécutables (COM et EXE)

Reprenons le cas de la section 6.3.1.3. avec le deuxième type de représentation.

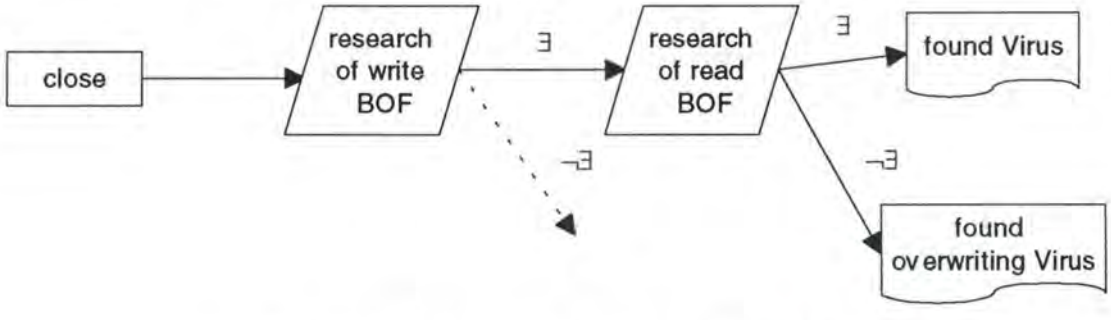


Fig. 6.4 : Infection de fichiers exécutables

La fonction initialisatrice de cette règle générique est la fonction 'Close' car une écriture dans un fichier n'est pas prise en compte par le DOS tant que le fichier n'a pas été fermé. En réalité, on ne peut dire dans quel état se trouve un fichier après écriture lorsqu'il n'a pas encore été fermé. Pour que le virus ait bien infecté le fichier, il a donc tout intérêt à fermer celui-ci correctement.
La règle déclenche alors un traitement de recherche d'écriture en début de fichier.

Si cette écriture n'a pas lieu, la règle est désactivée. On suppose en effet ici que le virus, pour prendre la main lors de l'exécution du fichier infecté, transforme l'en-tête du fichier pour que son code soit exécuté le premier. Après son exécution, le virus rend la main au programme infecté pour son exécution normale.
Si, par contre, l'écriture en début de fichier a bien lieu, un deuxième traitement est déclenché pour rechercher une lecture préalable du début de fichier. Si cette lecture n'a pas lieu, on est sûr d'être face à un virus réinscripteur car le début du fichier original a bel et bien été écrasé. En effet, le virus n'a su à aucun moment garder en mémoire l'en-tête du programme.
Si la lecture a lieu, on est un peu moins précis dans le message envoyé à l'utilisateur en disant simplement que l'on est face à un virus.

6.3.1.5.2 Création de fichiers exécutables

On recherche principalement dans ce schéma la création de virus compagnons. Ces virus utilisent l'ordre de priorité des extensions du DOS (voir 2.3.5.) en créant pour un fichier d'extension EXE un nouveau fichier de même nom mais avec une extension prioritaire (c'est le cas de l'extension COM). Ainsi, lorsque l'utilisateur croit exécuter un fichier, le virus est exécuté d'abord car son extension lui donne priorité. Le virus exécute ensuite ce fichier et l'utilisateur n'a rien vu.

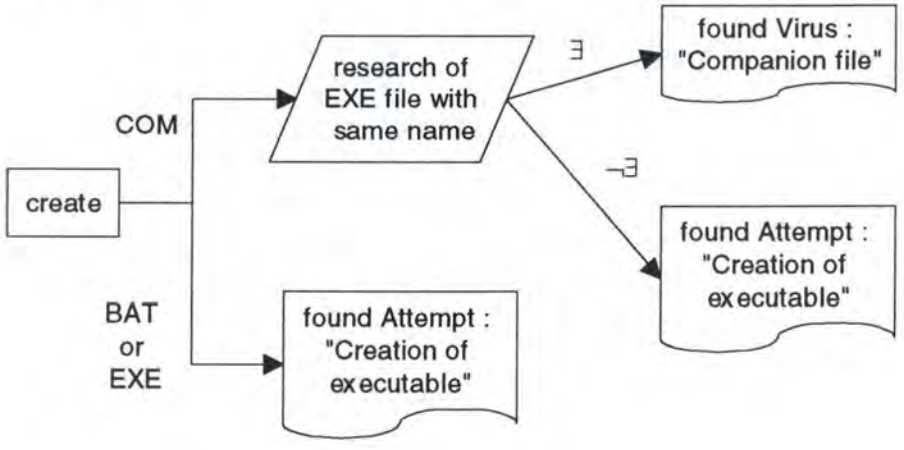


Fig. 6.5 : Création de fichiers exécutables

Lors de la création d'un fichier exécutable :

- si ce fichier est un fichier d'extension COM, la règle déclenche un premier traitement de recherche du fichier EXE de même nom. Le résultat de cette recherche se ressent dans le type de message envoyé à l'utilisateur. En effet, si ce fichier EXE existe, on est face à la création d'un fichier compagnon qui aura la main avant ce fichier EXE. Si ce fichier n'existe pas, on n'est pas sûr d'être face à un virus, alors on envoie à l'utilisateur un message d'alarme pour l'avertir d'un acte qui pourrait être important pour lui sans que l'on soit sûr d'être face à un virus.
- si, par contre, on est face à la création d'un fichier d'extension BAT ou EXE, on n'est pas nécessairement face à un virus, mais on envoie le même message d'alarme à l'utilisateur.

6.3.1.5.3 Renommage de fichier

D'autres virus utilisent une manière détournée pour infecter un fichier. Ils lisent d'abord un fichier, écrivent ce même fichier et leur propre corps dans un autre fichier temporaire. Ensuite,

ils effacent le fichier original et le remplacent par le fichier temporaire qu'ils ont créé par le biais du renommage.

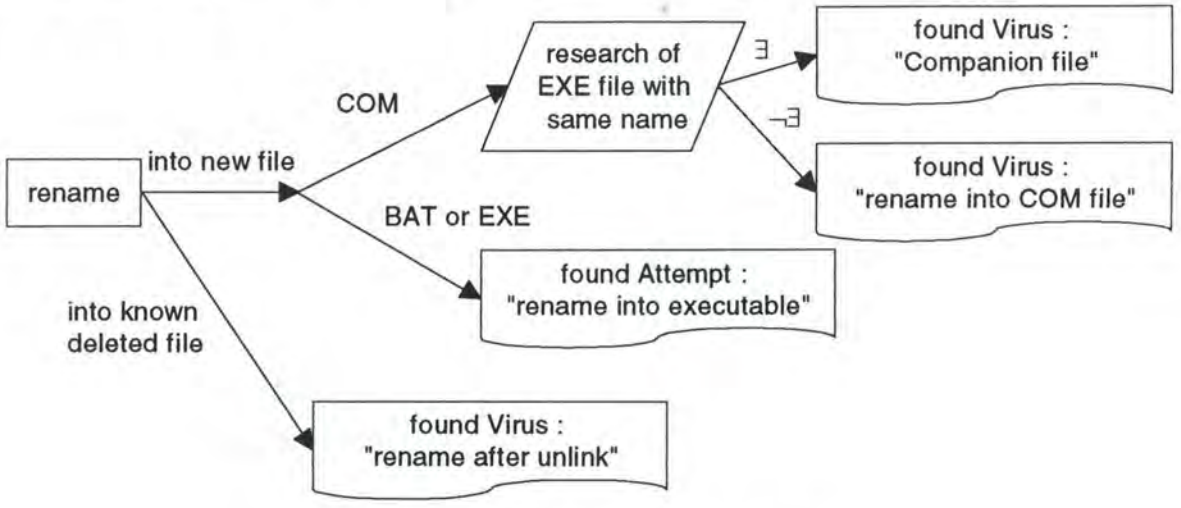


Fig. 6.6 : Renommage de fichier

Si la fonction initialisatrice est la fonction *rename* :

- si le nouveau nom du fichier est l'ancien d'un fichier qui a été préalablement effacé, nous sommes face à un virus.
- si le nom est nouveau, on différencie deux cas :
  - si on renomme dans un fichier BAT ou EXE, il y a envoi d'un message d'alarme à l'utilisateur ;
  - si on renomme dans un fichier COM, la règle déclenche un traitement de recherche d'un EXE ayant le même nom et si ce fichier EXE existe, il y a création d'un fichier compagnon par le mécanisme du renommage. On est alors en face d'un virus compagnon. Sinon, il y a envoi d'un message d'alarme à l'utilisateur.

Il faut savoir que, dans le DOS, on ne peut pas renommer un fichier avec le nom d'un fichier déjà existant, c'est pour cela que la première séparation (nouveau fichier ou fichier connu mais effacé) a été créée.

La technique de création d'un fichier compagnon par le renommage aurait pu déjouer la règle précédente de création de fichiers exécutables. On voit que dans notre cas, un ensemble d'actions légitimes paraissant tout à fait anodines au départ, quand elles sont mises en relation, prend tout son sens de malice.

6.3.2 Règles spécifiques au virus

Comme on a pu le voir dans la section précédente, on peut détecter la plupart des virus avec seulement quelques règles. D'un autre côté, un virus qui utilise une stratégie d'attaque inconnue ne sera pas détecté. Pour cette raison, on développe alors des règles spécifiques au virus.

Il faut de toutes manières commencer par une analyse statique du fichier audit, ainsi, on arrive à se faire une idée de l'attaque du virus en analysant les fonctions qu'il déclenche et sa stratégie.

Durant l'analyse des fonctions déclenchées, il est possible de faire un diagramme de contrôle de flux des fonctions utilisées par le virus pour infecter le fichier (car il ne faut pas oublier que c'est l'infection que l'on recherche depuis le début).

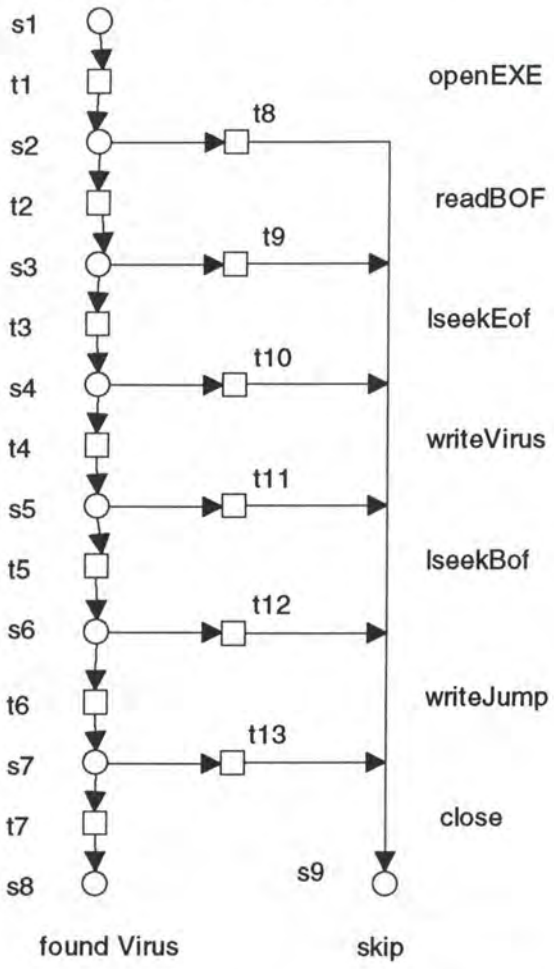


Fig. 6.7 : Diagramme de flux pour le virus Vienna

Voici le résultat que l'on peut obtenir en analysant le virus Vienna, ou plutôt en analysant son fichier audit d'exécution pour y voir sa technique d'attaque. On obtient alors ce type de graphique reprenant le modèle des diagrammes de transition d'états : sur la première colonne, on peut voir la séquence des différents états reliés entre eux par des actions menant à l'assurance de l'identification de l'attaque par le virus Vienna.
Sur la seconde colonne, de t8 à t13, on peut voir les différentes actions qui amènent à sortir de la règle.
Enfin, la dernière colonne reprend les différentes instructions que le virus Vienna utilise pour sa stratégie d'attaque :

- t1 est une action d'ouverture de fichier EXE (openEXE),
- t2 est une action de lecture de début de fichier (readBOF),
- t3 est une action de déplacement de pointeur en fin de fichier (lseekEOF),
- t4 est une action d'écriture (du virus en l'occurrence, write Virus),
- t5 est une action de déplacement de pointeur en début de fichier (lseekBOF),
- t6 est une action d'écriture en début de fichier (write Jump),
- et t7 est l'action de fermeture du fichier (close).

Donc, par la modélisation en diagramme de transitions d'états, on arrive à créer la règle spécifique au virus. Bien évidemment, cette règle reste rarement seule. Dès lors, on essaie de généraliser cette règle le plus possible pour atteindre une nouvelle règle générique permettant de détecter toute une classe de virus. Les règles spécifiques au virus sont plutôt la première étape du passage d'une analyse statique brute du fichier audit reprenant l'exécution du virus à la règle générique.

D'un autre côté, il arrive qu'il y ait des exceptions aux règles génériques, c'est également là qu'interviennent les règles spécifiques au virus ; règles qui s'occupent alors de détecter ce ou ces quelques virus non détectés par la règle générique.

6.3.3 Autres règles

Ces autres règles, comme cela a déjà été dit auparavant, servent à glaner de l'information à propos des virus pour permettre une classification plus facile. De plus, elles permettent de se rendre compte qu'il y a peut-être une infection quand des changements de date ou de taille arrivent mais qu'aucune règle générique et/ou spécifique ne se rende compte de l'infection. Cela permet ainsi de se rendre compte qu'on est peut-être face à une nouvelle stratégie d'infection.

Nous allons présenter les différentes caractéristiques pertinentes que l'on peut rechercher pour classifier les virus. Nous utiliserons également la méthode des diagrammes de transitions pour représenter les différentes règles.

6.3.3.1 Modification des attributs

Dans le DOS, chaque fichier possède des attributs obligatoires qui permettent de déterminer à quel type de fichier on a affaire. Pour retrouver les tentatives d'infection, il est parfois utile de rechercher les modifications d'attributs.
Il y a quatre attributs possibles : S - H - R - A.

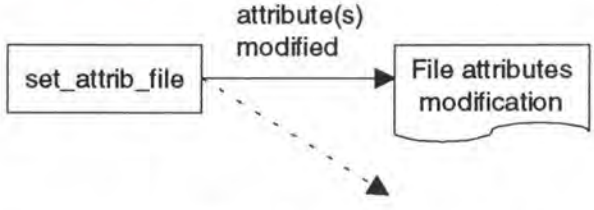


Fig. 6.8 : Modification d'attributs

6.3.3.1.1 S(ystem)

L'attribut S est réservé aux fichiers exécutables et de chargement du DOS. Cet attribut signale que le fichier ne peut ni être déplacé ni être copié, son adresse physique sur le disque étant primordiale.
La suppression ou l'apparition d'attribut S doit être considérée comme suspecte. Il y a certes des exceptions ; lors de l'installation de certains programmes protégés, il y a création de fichiers avec cet attribut contenant le numéro de licence ou un compteur pour mémoriser le nombre d'installations successives. Ici encore, le nom du commanditaire de l'opération est primordial et permettra de déterminer si l'opération est légitime ou non.

6.3.3.1.2 H(ide)

L'attribut H permet de ne pas faire apparaître le nom d'un fichier lors du listage des noms de fichiers d'un répertoire. De nos jours, tous les gestionnaires de fichiers montrent les fichiers

cachés. Si la modification a été demandée par l'intermédiaire du programme ATTRIB ou un gestionnaire de fichiers, il y a beaucoup de chances que ce soit l'utilisateur qui l'ait sciemment demandée.
Il est ici très difficile de se prononcer quant au danger que pourrait représenter le changement de cet attribut, mais l'opération de modification doit être notifiée.

6.3.3.1.3 R(ead only)

L'attribut R indique que le fichier possédant cet attribut est accessible en lecture seulement. Un exécutable possédant cet attribut peut être lu ou exécuté mais pas modifié.
En général, la modification de cet attribut est faite à la demande de l'utilisateur au moyen du programme ATTRIB du DOS.
Une séquence suspecte pourrait être par exemple la suppression de l'attribut R, la modification de l'exécutable concerné et la remise finale de l'attribut R. La plupart utilise ce type de modification en retirant l'attribut R des fichiers exécutables.

6.3.3.1.4 A(rchive)

L'attribut A indique que le fichier possédant cet attribut, a été modifié depuis la dernière sauvegarde. Ici, de nouveau, le commanditaire de l'opération de modification est très important. Si la demande vient de BACKUP, de ATTRIB ou d'un autre utilitaire de sauvegarde, il y a beaucoup de chances que l'opération soit tout à fait normale.
Par contre, un virus pourrait utiliser cet attribut comme marqueur, pour reconnaître si le fichier est déjà contaminé ou non.

6.3.3.2 Modification de la date et de l'heure

Lors d'une modification d'un fichier, normalement, la date et l'heure associées à ce fichier sont modifiées. La date et l'heure permettent alors de déterminer le moment de la dernière modification.
Pour un virus, il est dès lors primordial que l'on ne puisse pas voir qu'il a infecté un fichier. Donc, avant infection, il sauvegarde la date et l'heure associées au fichier, et après infection, il efface les traces de son infection en remettant la date et l'heure qu'il avait sauvées précédemment.
Cette modification de la date et de l'heure peut se faire grâce à l'utilisation d'une interruption. Il faut donc signaler toute demande d'appel de cette interruption.

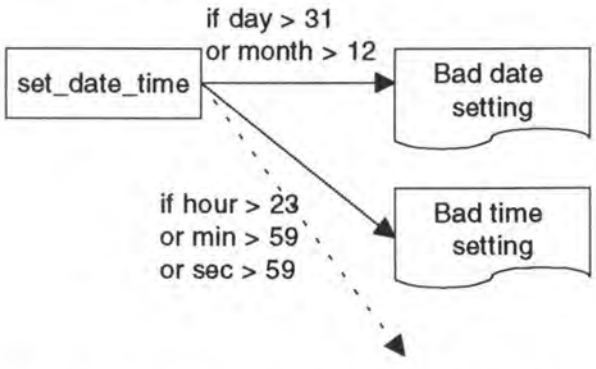


Fig. 6.9 : Modification de la date et/ou de l'heure

De plus, certains virus utilisent la date et l'heure pour marquer le fait qu'un fichier est infecté ou pas. Par exemple, le virus 4096 change le siècle dans la date, ce qui passe inaperçu pour l'utilisateur puisque lors d'un listage de fichiers d'un répertoire, seuls les deux derniers chiffres

de l'année apparaissent. Il est donc impossible de distinguer 1996 de 2096. Un autre virus utilise les millièmes de secondes.
Il est donc important de tester la validité de la date.

6.3.3.3 Modification de la taille d'un exécutable

Les premiers virus, en s'accrochant aux exécutables, faisaient grandir la taille de ceux-ci. Maintenant, certains virus utilisent un algorithme de compression pour réduire leur taille et celle du programme hôte, rendant alors impossible la détection de modification de taille par le virus.
Il est donc également important de signaler les modifications de tailles.

6.3.3.4 Exécution d'un fichier infecté

Il peut également être important de détecter si un fichier infecté est exécuté par le virus ou non. Cela permet d'avoir une caractéristique de plus du virus.

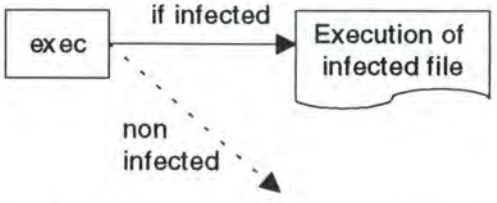


Fig. 6.10 : Exécution d'un fichier infecté

Par exemple, il est clair que lors de la création par le virus d'un fichier compagnon pour l'exécutable que le virus infecte, le virus doit exécuter ultérieurement l'exécutable que le virus voulait infecter, rendant ainsi la main au programme hôte.

7. Méthodologie de développement de règles en RUSSEL

Dans ce chapitre, nous allons montrer comment le langage RUSSEL peut être effectivement utilisé pour détecter un scénario d'infection. Comme montré dans le chapitre précédent, nous modélisons l'infection par soit un diagramme de transitions d'états, soit un diagramme des traitements, et nous allons montrer comment ces deux types de diagrammes peuvent être traduits en règles RUSSEL. Nous présenterons enfin l'implémentation que nous avons réalisée durant notre stage à Hambourg de la détection de virus via la modélisation par des diagrammes de traitements.

7.1 Développement de règles à partir de diagrammes de transitions d'états

7.1.1 Méthode de développement

Reprenons la structure générale d'un diagramme de transitions d'états du chapitre précédent.

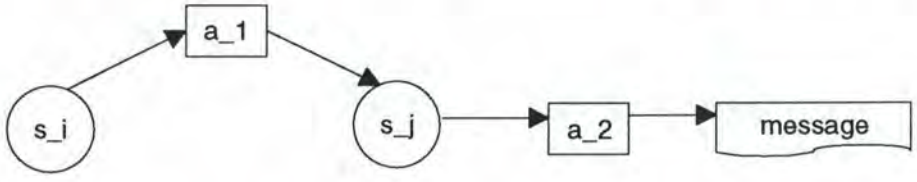


Fig. 7.1 : Structure générale d'un diagramme de transitions d'états

1. Chaque état dans le diagramme est représenté par une règle qui décrit l'état courant.
2. Une transition dans le diagramme est représentée par le mécanisme de déclenchement de règle du langage RUSSEL décrit dans le chapitre 4. La règle déclenchée représente le nouvel état courant dans le diagramme de transitions.
3. Les paramètres de la règle courante contiennent toute l'information pertinente collectée dans les états visités précédemment. Ils représentent la précondition de la règle déclenchée.
4. Les valeurs que l'on a donc pour une règle courante sont : les paramètres de la règle courante et les données contenues dans l'enregistrement courant du fichier audit que l'on analyse.
5. En particulier, la première règle active au début de processus de détection n'a pas de paramètres car il n'y a pas d'information contenue dans l'état initial. On n'a pas encore eu besoin de garder des informations sur le passé puisqu'il n'existe pas encore. On pourrait cependant dire que l'état initial contient une liste vide de paramètres.

7.1.2 Exemple

Reprenons une partie de la règle spécifique au virus Vienna qui utilisait la représentation en diagramme de transition d'états.

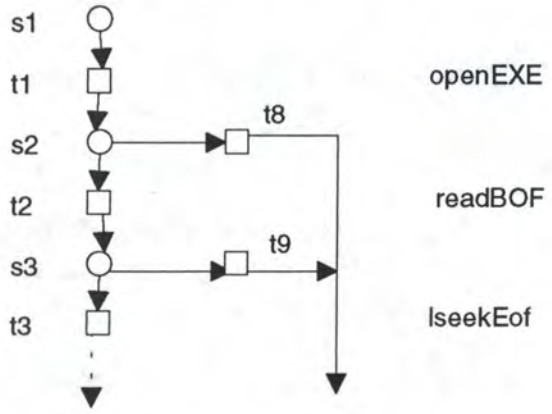


Fig. 7.2 : Diagramme partiel des flux pour le virus Vienna

Dans l'exemple que nous allons développer, nous allons de nouveau utiliser les mêmes notations pour représenter les règles RUSSEL :

- les mots-clés tirés de la grammaire BNF de RUSSEL sont en gras : **if** ... **fi**
- les champs de l'enregistrement courant (présentés en annexe 3) sont en italique : *CS*
- les paramètres de la règle sont en Courier: `filename`

A la fin du diagramme de transitions d'états se trouve l'état final qui ne déclenche pas de règle supplémentaire, mais à la place, il lance une procédure qui envoie un message d'alarme décrivant l'infection et utilisant les données accumulées tout le long de la séquence d'états préalablement visités.
Nous allons reprendre dans notre exemple les cinq points développés dans la section 7.1.1 pour les illustrer chacun.

Commençons par la règle initiale, celle décrite dans le point 5, règle qui n'a donc aucun paramètre (ou une liste vide de paramètres).
Voici son implémentation :

```
rule openEXE ;

begin
if      strToInt(Function) = 45                 /* on est face à la fonction open */
        and match('.*COM$', arg_str1) = 1       /* et c'est un fichier COM */
        → trigger off for_next readBOF(ret_AX, CS, arg_str1)
                        /* déclenchement pour l'enregistrement suivant de la règle readBOF
                                comme paramètres :
                                        • le handle du fichier,
                                        • le codesegment,
                                        • le nom du fichier.
                        */
fi ;
trigger off for_next  openEXE
end ;
```

Explication :

- la règle vérifie la première transition, i.e. est-ce bien une ouverture de fichier et est-on face à un fichier COM ? Si c'est le cas, on peut alors passer en état deux par l'appel à la seconde règle qui est readBOF.
  On garde dans cet appel tous les paramètres importants pour la règle suivante. Dans readBOF, nous aurons besoin pour pouvoir passer à l'état suivant :
  - du handle car les fonctions DOS l'utilisent pour identifier le fichier qui a été ouvert,
  - du codesegment qui indique l'endroit du programme en mémoire d'où vient l'appel,
  - et enfin, du nom du fichier, qui ne sera pas utilisé avant l'affichage final du message. Il doit donc être passé comme paramètre dans la règle.
- Ensuite, la règle se redéclenche de nouveau pour permettre une analyse d'une seconde infection éventuelle durant le premier processus d'infection. Ainsi, aucune infection, même si elle se déroule durant une autre infection, n'échappera à la détection spécifique du virus Vienna. On a alors une détection simultanée de plusieurs infections.

Développons maintenant la seconde règle pour pouvoir illustrer les quatre autres points.

```
rule readBOF (handle, codeSeg, filename : string) ;

begin
if
        strToInt(Function) = 47                 /* Fonction Read */
    and arg_BX = handle                         /* il s'agit du même fichier */
    and CS = codeSeg                            /* lu par le même processus l'ayant
                                                   ouvert */
    → trigger off for_next lseekEOF (handle, codeSeg, arg_CX, filename) ;

        strToInt(Function) = 46                 /* Fonction close */
    and arg_BX = handle
    and CS = codeSeg
    → skip ;

    true
    → trigger off for_next readBOF(handle, codeSeg, filename)
fi
end ;
```

Explication :

- la première condition que le **if** vérifie est : est-on face à la transition suivante dans la séquence que l'on désire retrouver pour détecter le virus Vienna ? Si oui, on déclenche la règle suivante lseekEOF qui représentera le nouvel état courant. On garde dans cet appel :
  - les mêmes paramètres que précédemment car on en aura à chaque fois besoin ;
  - et en plus le nombre de bytes lus dans le fichier.
- sinon, la seconde condition vérifie : ferme-t-on le fichier ? qui est la condition de sortie sans message d'alarme du processus de détection du virus Vienna. Le **skip** est utilisé pour exprimer le fait que l'on ne veut rien faire, donc sortir du processus en ne déclenchant plus aucune autre règle.

- sinon, la condition par défaut (**true**) relance la recherche d'une autre lecture en début de fichier (readBOF) pour ce même fichier pour continuer la séquence du processus de détection du virus Vienna.

Par rapport à la méthode de développement décrite à la section 7.1.1., le déclenchement de la règle readBOF dans la règle précédente (openEXE) est l'illustration du point 2. En effet, son déclenchement représente la transition sur le diagramme de transitions d'états et la règle readBOF à l'étape suivante représente bien le nouvel état courant.
On peut également voir dans cet exemple, que la règle courante représente bien l'état courant (illustration du point 1) grâce à ses paramètres et les valeurs des champs de l'enregistrement courant (illustration du point 4).
Pour illustrer le point 3, on peut enfin dire que l'information collectée auparavant (représentée par les paramètres de la règle) est bien pertinente : en effet, le handle du fichier et le codesegment sont utiles pour s'assurer la bonne identification du fichier dont on pense qu'il est peut-être attaqué par le virus Vienna ; et le nom du fichier est gardé pour le message final.
Pour être précis dans notre exemple, la règle finale du processus représentant la fermeture finale du fichier attaqué contient l'appel de procédure prédéfinie suivant :

println('Virus Vienna found infecting file ', `filename`) ;

Pour terminer l'illustration, il ne faut pas oublier que, dans le cas d'une règle spécifique au virus (qui peut alors être représentée par un programme complet d'analyse), il y a une règle qui déclenche le démarrage du processus de détection :

init_action ;
begin
    trigger off for_next openEXE
end.

7.2 Développement de règles à partir de diagrammes des traitements

7.2.1 Méthode de développement

Réutilisons la structure des diagrammes des traitements du chapitre précédent.

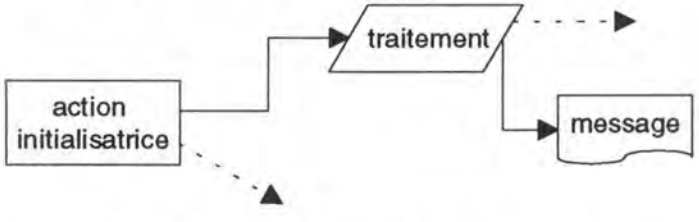


Fig. 7.3 : Diagramme des traitements

On suppose dans ce type de diagramme que l'on peut faire des traitements de recherches sur des données passées. Pratiquement, il est évidemment impossible d'être aussi parfait, mais il y a déjà moyen d'aller assez profondément dans le contenu des informations que l'on garde du passé. Pour réaliser l'emmagasinage des des informations importantes pour les recherches, on effectue un stockage de l'information *bottom-up* car il se réalise dans le cheminement chronologique de l'infection.

Du point de vue programmation, pour pouvoir en garder la plus grande quantité, il suffit d'utiliser les tables associatives que nous avons déjà présentées dans le chapitre 5.
Avec ces tables, il est possible de tout garder. Seulement, pour ne pas alourdir le système et n'utiliser que les informations pertinentes, vu que l'on ne va pas utiliser tout ce que l'on peut savoir du passé, il est fortement conseillé de décider auparavant quelles informations sont pertinentes (en termes d'utilisation durant le programme d'analyse du fichier audit) et celles qui ne seront jamais utilisées.

Il est vrai que l'on pourrait utiliser le système de la méthode précédente en gardant tout dans une liste de paramètres ou dans plusieurs règles avec quelques paramètres. Cela deviendrait alors fastidieux mais tout à fait possible à réaliser.
Il suffit de prendre l'exemple d'un virus compagnon. Lorsqu'il commence son processus d'infection, il recherche un fichier EXE dont il va créer le fichier compagnon. Dans ce cas, cela supposerait donc que l'on devrait pouvoir garder comme paramètres tous les fichiers EXE contenus dans le système.

L'action initialisatrice du diagramme est représentée par une règle RUSSEL de mise à jour de notre connaissance sur le passé grâce au nouvel apport d'informations créés par le nouvel enregistrement courant. On retire donc dans ce cas, au niveau de l'implémentation, l'information que l'on considère comme pertinente dans l'enregistrement courant et on l'installe ou on la met à jour dans les tables associatives. Cette règle lance également le traitement s'il y en a un.

Le traitement est, du point de vue programmation, une règle déclenchée par la règle expliquée au paragraphe précédent (déclenchement exprimé par l'arc de transition sur le diagramme). Elle permet la recherche de données du passé dont on a besoin et le traitement du résultat de ces recherches, suivi parfois d'une dernière mise à jour de la connaissance sur le passé.
Il est en effet possible que certaines données ne puissent être retirées des données pertinentes du passé tant que l'on n'a pas fini le traitement des résultats de la recherche. Un exemple reprenant cette dernière mise à jour est développé au point 7.2.2.4.

Les traitements déclenchés par d'autres traitements dans le diagramme peuvent être implémentés de manières différentes :

- on peut rassembler ces différents traitements en une seule règle qui reprend alors la séquence des différents traitements du diagramme ;
- on peut également les séparer dans des règles distinctes. Cela permet alors la réutilisation de traitements spécifiques par d'autres traitements.

Il est évident que les commentaires utilisés sur les différents arcs du diagramme sont en fait représentés, du point de vue implémentation, par les conditions de déclenchement des traitements ou des envois de messages.

7.2.2 Exemple : le programme du stage

Pour illustrer la méthode décrite ci-dessus, nous allons expliquer l'implémentation du programme d'analyse que nous avons créé durant notre stage à Hambourg. Ce programme permet la détection d'infection par virus à partir de règles génériques, plus une utilisation de certaines autres règles, recherchant des caractéristiques des virus infectant le fichier.
Le programme, lors de l'analyse du fichier audit-trail reprenant l'exécution d'un fichier suspect, permet de détecter si le système émulé subit une infection par un virus, permettant alors d'affirmer que ce fichier suspect contient un virus. De plus, le programme permet de

rechercher des caractéristiques de ce virus lorsqu'on s'est rendu compte de sa présence dans l'émulation. Le programme envoie alors à l'utilisateur des messages pour lui relater l'historique du processus d'infection du virus et les caractéristiques pertinentes pour l'identifier.
Le programme permet à l'utilisateur de mieux cerner le virus et de le classifier plus facilement.

7.2.2.1 Les tables associatives

Nous allons d'abord citer les différentes informations dont nous avons besoin à propos du passé et que nous sommes sûrs d'utiliser lors de l'exécution du programme d'analyse.

- Nous gardions des informations sur les fichiers (table dynamique) :
  - *filename* : nom du fichier avec le chemin courant (identifiant) et le drive dans lequel il se trouve ;
  - *unlink* : booléen représentant le fait que le fichier a été effacé ou pas ;
  - *rename* : booléen représentant le fait que le fichier a été renommé ou pas ;
  - *writeBOF* : booléen représentant le fait que l'on a écrit au début du fichier ;
  - *filelength* : longueur connue du fichier ;
  - *surelength* : booléen représentant le fait que l'on est sûr ou pas de la longueur connue du fichier ;
  - *attributes* : attributs des fichiers (sous forme d'entier) ;
  - *filedate* : dernière date connue du fichier ;
  - *filetime* : dernier temps connu du fichier ;
  - *lastfileposition* : dernière position du pointeur sur le fichier ;
  - *name of the replaced file*: nom du fichier qui a été remplacé par le fichier considéré ;
  - *file created* : fichier créé ou non ;
  - *file infected* : fichier infecté ou non.
- Nous gardions des informations sur les handle des fichiers (table dynamique):
  - *file handle* : numéro d'identification du fichier qu'utilisent certaines fonctions DOS ;
  - *file name* : nom du fichier correspondant au handle.
- Nous gardions des informations sur les drives (table dynamique) :
  - *drive letter* : lettre des disques considérés ;
  - *current path* : chemin courant dans le drive considéré.
- Nous gardions des informations sur les fichiers avant exécution du virus (table statique) :
  - *file number* : numéro d'indice du fichier dans la table ;
  - *filename* : nom du fichier ;
  - *original file size* : taille originale du fichier ;
  - *original file date* : date originale du fichier ;
  - *original file time* : temps original du fichier.
- Une dernière table statique avait été créée reprenant tous les noms des fichiers connus avant exécution du virus. Cette table a été utilisée pour la recherche de fichiers compagnons et pour pouvoir comparer à la fin de l'analyse si le virus avait créé durant son exécution de nouveaux fichiers.

7.2.2.2 Le programme SEEK

Dans l'émulation DOS utilisée, avant de lancer l'exécution du virus pour l'auditer, nous lancions un programme DOS de nom SEEK qui permettait de trouver toutes les informations sur les fichiers quand le système émulé est encore exempt de tout virus. L'exécution de ce programme se trouvant en premier lieu dans l'audit-trail nous permettait de remplir la table statique d'informations sur les fichiers existants.

Cette table nous était fortement utile pour la recherche des caractéristiques des virus. Grâce à cette table, à la fin du fichier, il nous suffisait de comparer la table statique d'informations sur les fichiers et la table dynamique d'informations sur les fichiers transformée durant l'analyse de l'audit de l'exécution du virus pour pouvoir trouver les changements de taille, de date et de temps.
Cette comparaison était effectuée à la fin du programme d'analyse quand tout le fichier audit avait été analysé. On était alors sûr que le virus avait terminé son infection.

Expliquons la règle initiatrice de tout programme ASAX : **init_action**

```
init_action ;
begin
        counter := 0 ;
        trigger off for_next installTables ;
        trigger off at_completion statistics
end.
```

Dans notre cas, cette règle :

- démarre un compteur : lors de l'impression d'un message à l'écran, cela permet à l'utilisateur qui le désire d'aller rechercher la cause du message dans le fichier d'audit par lui-même, et de savoir quelle instruction a provoqué ce message ;
- lance ensuite pour le premier enregistrement courant une règle *installTables* qui initialise les tables associatives dans la mémoire. Cette règle lance alors la règle générale du programme *recordAnalysis* qui sera développée plus tard ;
- permet d'activer, pour la fin de l'analyse, une règle de statistiques :
    - impression de la table dynamique d'informations sur les fichiers ;
    - comparaison pour tous les fichiers existants d'un possible changement de taille, de date et/ou de temps ;
    - libération de la mémoire occupée par les tables associatives.

Il est à remarquer que la règle **init_action** de déclenchement du programme d'analyse déclenche souvent comme première règle, un premier remplissage des tables associatives car il se peut que dès le début, on ait besoin de données dès le premier enregistrement courant.

7.2.2.3 Action initialisatrice du diagramme

Pour l'implémentation des actions initialisatrices des différents diagrammes de traitements, nous avons intégré toutes les règles génériques et les autres règles de recherche de caractéristiques des virus en deux règles RUSSEL.

La première règle s'appelle *recordAnalysis* : ce nom s'explique par le fait que l'on commence par une analyse de l'enregistrement courant en utilisant les informations qui y sont contenues. Notre analyse de l'enregistrement se passe en général de cette manière :

- détermination de la fonction DOS appelée ;
- détermination du résultat fourni par la fonction DOS, i.e. on regarde si la fonction DOS s'est exécutée normalement et n'a pas fourni de message d'erreur ;
- ensuite, on range dans l'ensemble des règles actives pour l'enregistrement courant (**trigger off for_current**) la règle de déclenchement de règles (voir juste après) ;
- ensuite, elle se relance pour l'enregistrement suivant (**trigger off for_next**)

Une analyse particulière est réalisée par *recordAnalysis*, c'est dans le cas de l'analyse du résultat fourni par le programme SEEK dans l'émulation du système. A ce moment-là, une règle *fill_t45* est lancée. Cette règle remplit les tables statiques d'informations sur les fichiers avant exécution du virus et de noms des fichiers connus. Cette règle se relance pour les enregistrements suivants jusqu'à ce que l'on atteigne la fin de l'exécution du programme SEEK dans le fichier audit. Alors, cette règle relance *recordAnalysis* qui ne lancera plus jamais *fill_t45*.

Voici le corps allégé de la règle *recordAnalysis* :

```
/* ************************************************** */
rule recordAnalysis;
/*      this rule detects if the codesegment belongs to the virus
        and triggers off ruleTrigger
*/
var c, Fn, handle, handle_good : integer;
begin

/* see if the handle of the file is higher than 4 (because between 1 and 4 : I/O system) then the
handle_good is equal to 1 (we moved the code but you can find it in the Appendix 1) */

if handle_good = 1
        -->
                /* we only consider the functions who have their ret_CF = 0 , i.e.
                   the function call was successful */

                if ((Fn = 44 or Fn = 81 or Fn = 45 or Fn = 46 or Fn = 47 or Fn = 48
                   or Fn = 49 or Fn = 50 or (Fn = 51 and strToInt(arg_AL) = 0) or Fn = 53
                   or Fn = 54 or Fn = 59 or Fn = 72 or Fn = 73 or Fn = 74 or Fn = 96)
                   and strToInt(ret_CF) = 0)
                        --> begin
                                trigger off for_current ruleTrigger;
                                trigger off for_next recordAnalysis
                            end;

                   (Fn = 3 or Fn = 13 or (Fn = 43 and strToInt(ret_CF)=0))
                        --> begin
                                trigger off for_current ruleTrigger;
                                trigger off for_next recordAnalysis
                            end;

/*particular case */(Fn = 64 and arg_str1 = 'A:\*.*')
                        --> trigger off for_current fill_t45;
                   true
                        --> trigger off for_next recordAnalysis
                fi;
   true --> trigger off for_next recordAnalysis
  fi
end;
/* ************************************************** */
```

La seconde règle *ruleTrigger* déclenche la règle adéquate pour la mise à jour des tables associatives par rapport à la fonction DOS appelée.

Voici son corps allégé :

```
/* ************************************************** */
rule ruleTrigger;
var Fn: integer;
/*      this rule triggers the good rule for the updating of the associative tables
        regarding to the operation
*/
begin
  Fn := strToInt(Function);

  if    Fn =  3  --> trigger off for_current setDrive;
        Fn = 13  --> trigger off for_current getDefaultDrive;
        Fn = 43  --> trigger off for_current changeDir;
        Fn = 44  --> trigger off for_current createFile;
        Fn = 45  --> trigger off for_current openFile;
        Fn = 46  --> trigger off for_current closeFile;
        Fn = 47  --> trigger off for_current readFile;
        Fn = 48  --> trigger off for_current writeFile;
        Fn = 49  --> trigger off for_current unlinkFile;
        Fn = 50  --> trigger off for_current lseekFile;
        Fn = 51  --> trigger off for_current getSetAttribFile;
        Fn = 53  --> trigger off for_current duplicate;
        Fn = 54  --> trigger off for_current forceDuplicate;
        Fn = 59  --> trigger off for_current execFile;
        Fn = 72  --> trigger off for_current renameFile;
        Fn = 73  --> trigger off for_current getDateTimeFile;
        Fn = 74  --> trigger off for_current setDateTimeFile;
        Fn = 81  --> trigger off for_current createFile;
        Fn = 96  --> trigger off for_current extendedOpenFile
fi
end;
/* ************************************************** */
```

Cette règle déclenchée porte le nom de la fonction DOS. Nous nous sommes occupés des fonctions :

- setDrive, getDefaultDrive, changeDir : ces trois fonctions permettent la recherche du chemin courant pour le fichier considéré, la mise à jour des tables associatives concernant les informations sur les drives et la mise à jour de deux variables globales, une pour le drive actuel et l'autre pour le chemin actuel (drive inclus). Nous avons utilisé ce système pour permettre l'uniformisation des noms de fichiers.
- createFile, openFile, closeFile, readFile, writeFile, unlinkFile, lseekFile, getSetAttribFile, execFile, renameFile, getDateTimeFile, setDateTimeFile, extendedOpenFile : ces quelques fonctions habituelles sur les fichiers s'occupent de la mise à jour des tables associatives dynamiques.
- duplicate, forceDuplicate : ces deux fonctions permettaient la mise à jour de la table dynamique des handles de fichiers car elles permettent de créer un deuxième handle pour un même fichier.

Il est à remarquer que toute cette séquence de déclenchement de règles (recordAnalysis suivi de ruleTrigger suivi d'une des règles citées juste avant) se fait avec le mécanisme du **for_current** car toutes ces règles portent bien évidemment sur le même enregistrement dont le traitement n'est pas encore fini.

Développons la règle *rename* pour montrer le mécanisme de mise à jour de tables associatives. Supposons que la fonction soit du type : rename A B. A est l'ancien nom, B le nouveau nom. La règle :

- construit d'abord le nom complet du fichier pour conserver l'uniformité des noms de fichiers ;
- vérifie si l'ancien nom pour le renommage (A) existe déjà dans la table dynamique des noms de fichiers connus, sinon elle l'y installe ;
- va rechercher toutes les informations le concernant dans la table dynamique ;
- construit ensuite le nouveau nom complet du fichier (B) et fait la même vérification ;
- si ce nouveau fichier n'existe pas encore dans la table dynamique des informations sur les fichiers, on l'y installe et on supprime de la table dynamique l'ancien nom car cela signifie qu'on vient simplement de changer le nom du fichier en un nouveau nom qui n'a jamais existé auparavant ;
- s'il existait déjà, on met à jour un champ spécial (*name of the replaced file*) qui est d'habitude vide. Ce champ prend comme valeur le nom du fichier qui vient d'être remplacé. Cela est fait pour ne pas perdre la trace de ce renommage pour l'analyse qui va suivre dans le traitement.

```
/* ************************************************** */
rule renameFile;
/*      updates associative tables for the function 72 : rename
*/
var c1, c2,
   vtab, nbunlink, nbrename, nbwrite,vfile_length,
   vsure_length, vattrib, vlfp, val_create, val_infected : integer;
   val_key, vfile_date,vfile_time ,new_name, filename, filename2,strtemp : string;
begin
  if match('[A-Z]:\\',arg_str1) = 1
        --> filename := arg_str1;
    match('^\\',arg_str1) = 1
        --> begin
              strtemp := getSubStr(current_path,1,2);
              filename := concatStr(strtemp,arg_str1)
            end;
    true
        --> filename := concatStr(current_path,arg_str1)
  fi;
  filename := upper(filename);
  c1 := isMember(t5,filename);
  if c1 = 0
        --> skip;
    c1 = -4
        --> begin
              count_t4 := count_t4 + 1;
              c2 := install_g(t4,count_t4,filename,-1,'','');
              c2 := install_g(t5,filename,count_t4)
            end
  fi;
  val_key := filename;  /* research of all the values */
  c1 := retrieve_g(t1, val_key,nbunlink, nbrename, nbwrite,
   vfile_length, vsure_length, vattrib, vfile_date, vfile_time, vlfp, new_name,val_create,val_infected);
  if match('^[A-Z]:\\',arg_str2) = 1
        --> filename2 := arg_str2;
    match('^\\',arg_str2) = 1
        --> begin
              strtemp := getSubStr(current_path,1,2);
```

```
                filename2 := concatStr(strtemp,arg_str2)
            end;
    true
        --> filename2 := concatStr(current_path,arg_str2)
fi;
filename2 := upper(filename2);
c1 := isMember(t5,filename2);
if c1 = 0
        --> skip;
    c1 = -4
        --> begin
                count_t4 := count_t4 + 1;
                c2 := install_g(t4,count_t4,filename2,-1,'','');
                c2 := install_g(t5,filename2,count_t4)
            end
fi;
                /* if the command looks like : rename A B
                   if B already exists, A is put as parameter for the file B (parameter 14)
                   if B doesn't exist in the table, creation of a new entry with the old values
                     of A but with the name B */
c1 := isMember(t1, filename2);
if c1 = 0
        --> begin
                c2 := update(t1, filename2, 11, filename);
                c2 := update(t1, filename2, 3, 1)
            end;
    c1 = -4
        --> begin
                c2 := install_g(t1,filename2,0,1,0,
                        vfile_length, vsure_length,vattrib,vfile_date,vfile_time,0,'',0,val_infected);
                c2 := Remove(t1, filename)
            end
fi;
trigger off for_current postAnalysis
end;
/* ************************************************** */
```

7.2.2.4 Le(s) traitement(s)

Nous avons déjà dit auparavant que les traitements permettaient la recherche de données du passé et le traitement du résultat de cette recherche. Du point de vue implémentation, cela se traduit par la recherche des informations nécessaires dans les tables associatives.
Nous avons également dit que parfois, après ces traitements, il y avait une dernière mise à jour des tables associatives qui, si elle avait été faite auparavant, n'aurait pas permis l'analyse correcte du cas dans lequel on se trouvait.
Dans notre implémentation, nous avons décidé de rassembler tous les traitements en une seule règle qui se sépare en sous-traitements selon la fonction DOS appelée. Cette règle s'appelle *postAnalysis*.
C'est l'exemple de la règle générique commençant par l'action *rename* de renommage de fichiers que nous allons expliquer en profondeur. Les autres règles suivent le même cheminement de construction (souvent sans la dernière mise à jour).

Reprenons d'abord le schéma :

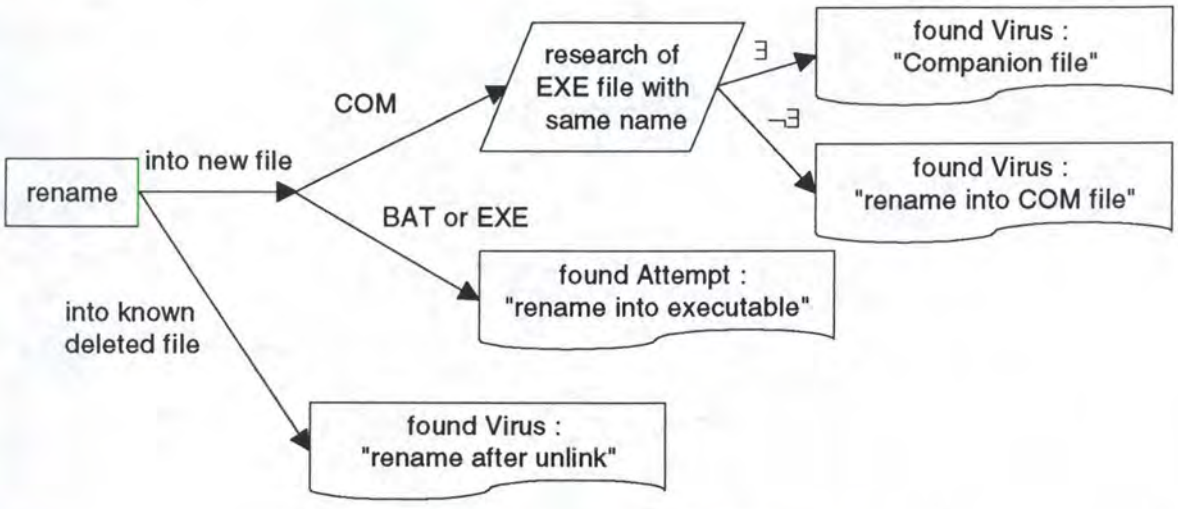


Fig. 7.4 : Renommage de fichiers

Nous avons expliqué dans le paragraphe précédent la mise à jour des tables pour cette fonction rename, nous allons maintenant expliquer son traitement par rapport au code :
Le traitement vérifie si le champ *name of the replaced file* est vide ou pas :

- s'il est vide, cela signifie que le renommage a lieu avec un fichier qui n'a jamais existé auparavant dans le système ;
    - On vérifie alors si c'est un fichier COM. Si c'est le cas, on lui recherche un fichier compagnon exécutable (EXE). Dans ce cas, on envoie un message du style : « Virus trouvé : tel fichier a été renommé en tant que fichier compagnon ». Sinon un autre message est envoyé : « Tentative malicieuse : renommage en fichier de commandes (COM) » ;
    - si le nouveau nom est un fichier EXE, on envoie un message : « Tentative malicieuse : renommage en fichier exécutable » ;
- sinon, cela signifie que ce fichier a déjà existé auparavant.
    - On recherche alors si cet ancien fichier a été effacé et si le nouveau fichier a été créé. Si c'est le cas, envoi d'un message : « Virus trouvé : un fichier a été renommé dans un nouveau nom qui a déjà existé et avait été effacé auparavant » . On vérifie que le nouveau fichier a été créé pour savoir s'il a bien été construit de toutes pièces.
    - Ensuite, vu que l'ancien fichier existe toujours pour nous, il faut le retirer de notre passé vu qu'il a un nouveau nom maintenant. On réalise cela en enlevant l'ancien fichier de la table associative dynamique des informations sur les fichiers tout en gardant ses champs et ensuite, on réinstalle le nouveau fichier avec les paramètres intéressants de l'ancien fichier en notant dans son dernier champ (*file infected*) qu'il a été évidemment infecté.

On peut voir dans le dernier point la dernière mise à jour de la table associative. C'est le seul cas dans notre programme où une dernière mise à jour est nécessaire après traitement.

Voici le corps de *postAnalysis* pour cette partie concernant le traitement de la règle déclenchée par la fonction *rename* :

```
/* ************************************************** */
rule postAnalysis;
/*      This rule analyses the associative tables after modification by the rule
        regarding the operation
*/
var     val_unlink, val_rename, val_write, val_file_length, val_sure_length, val_attrib, val_lfp,
        val_create, val_infected, Fn, c1, c2, handle, val_ind, long : integer;
        val_key, val_file_rename, val_file_rename2, val_file_date, val_file_time, filename,
        file_temp : string;
begin
  Fn := strToInt(Function);
/* research of the values of the file being considered */
.....
if      .....

        Fn = 72  /* rename */
    --> begin
                if match('[A-Z]:\\',arg_str2) = 1
                        --> filename := arg_str2;
                  true
                        --> filename := concatStr(current_path,arg_str2)
                fi;
                filename := upper(filename);
                val_key := filename;
                val_ind := 11;
                c1 := retrieve_1(t1,val_key,val_ind,val_file_rename);
                if val_file_rename = ''
                --> if match('.*COM$',val_key) = 1
                        --> begin
                                long := strLen(val_key);
                                file_temp := getSubStr(val_key,1,long - 3);
                                file_temp := concatStr(file_temp,'EXE');  /* if renaming in a COM file
                                                        and an EXE file with the same name already
                                                        exists, it may be a companion virus */
                                c1 := isMember(t5,file_temp);
                                if c1 = 0
                                --> trigger off for_current
                                        foundVirus(CS,filename,'renamed into Companion file');
                                  c1 = -4
                                --> trigger off for_current
                                        foundAttempt(CS,filename,'renamed into COM file')
                                fi
                            end;
                    match('.*EXE$',val_key) = 1 or match('.*BAT$',val_key) = 1
                                /* renaming in an EXE or BAT file (that didn't exist before) */
                        --> trigger off for_current
                                foundAttempt(CS,filename,'renamed into executable file')
                   fi;
                 true
                --> begin
                        val_ind := 2;
                        c1 := retrieve_1(t1,val_key,val_ind,val_unlink);
                        val_ind := 12;
                        c1 := retrieve_1(t1,val_file_rename,val_ind,val_create);
                        if val_unlink = 1
                        and val_create = 1
                        --> begin
                                trigger off for_current
                                        foundVirus(CS,filename,concatStr('renamed after
                                                        unlink by ',val_file_rename));
```

```
                                c1:= Remove(t1,val_key);
                                if match('[A-Z]:\\',arg_str1) = 1
                                --> filename := arg_str1;
                                  true
                                --> filename := concatStr(current_path,arg_str1)
                                fi;
                                filename := upper(filename);
                                if match('[A-Z]:\\',arg_str2) = 1
                                --> file_temp := arg_str2;
                                  true
                                --> file_temp := concatStr(current_path,arg_str2)
                                fi;             /* after having removed the old file, installation of
                                                   the new file with the parameters of the old file */
                                file_temp := upper(file_temp);
                                c1:=retrieve_g(t1,filename,val_unlink,val_rename,val_write,
                                        val_file_length,val_sure_length,val_attrib,val_file_date,
                                        val_file_time,val_lfp,val_file_rename2,val_create,
                                        val_infected);
                                c1:= install_g(t1,file_temp,0,0,0,val_file_length, val_sure_length,
                                        val_attrib,val_file_date,val_file_time,0,'',0,val_infected);
                                c1 := Remove(t1,filename)
                            end
                        fi
                    end
                fi
            end;
        Fn = 73  /* getDateTimeFile */
        --> skip;
.....
fi
end;
/* ************************************************** */
```

7.3 Avantages et désavantages

Quand il ne faut pas garder trop d'informations sur le passé dans la création d'une règle spécifique au virus, il est préférable d'utiliser la méthodes de diagramme de transitions d'états car cette méthode montre bien le cheminement que suit le virus et l'implémentation de cette règle, quand elle ne doit pas être intégrée avec d'autres règles, est immédiate.

Par contre, pour la représentation d'une règle générique, la méthode de diagramme des traitements est conseillée car :

- la plupart du temps, au moment de l'implémentation d'une règle générique représentée par un diagramme de transitions d'états, il faut introduire des conditions temporelles de sortie du schéma (timeout) pour éviter que la règle ne reste éternellement active en mémoire et ne tourne pour rien. On peut en effet affirmer qu'un virus ne met pas des heures pour infecter un fichier. La représentation par diagramme des traitements résout ce problème puisque la règle générique est déclenchée par l'action qui termine le diagramme de transitions d'états correspondant. On évite ainsi le risque d'explosion du nombre de règles actives en mémoire.
- un programme ASAX n'est pas définitif, ce qui est le cas des programmes d'analyse de détection d'infection par virus, puisque de nouvelles stratégies peuvent apparaître à tout moment, on ne peut jamais savoir l'information que l'on doit garder. Donc, il se peut que lorsqu'il faut intégrer une nouvelle règle générique dans le programme, il

faut étendre la liste de paramètres pour toutes les règles, tandis que par la gestion des tables associatives, ce problème est facilement résolu.

En conclusion, quand on sait parfaitement les limites du programme que l'on crée (programme définitif ou non) et que l'on ne doit pas garder trop d'informations du passé, la méthode des diagrammes de transitions d'états est conseillée.
Mais, dès qu'il faut garder beaucoup d'informations ou lorsqu'il y a beaucoup de dynamisme dans le programme (i.e. qu'il peut évoluer à tout moment par l'intégration d'une nouvelle règle par exemple), il est préférable d'utiliser la méthode des diagrammes des traitements.
Pour la représentation des autres règles, il est conseillé d'utiliser la méthode des diagrammes de traitements car ces règles sont souvent intégrées dans des programmes d'analyse contenant d'autres règles génériques pour permettre de se rendre compte que l'on est peut-être face à une exception. L'intégration sera alors plus facile.

8. Remarque sur la gestion de la mémoire en RUSSEL

Ce chapitre présente un aspect de la gestion de la mémoire de RUSSEL, en l'occurrence la gestion des variables. C'est lors de la manipulation de ces différents types de variables que nous avons rencontré une erreur dans l'implémentation de RUSSEL que nous tentons d'isoler et pour lequel nous proposons une correction. Il y aura également dans ce chapitre une petite section qui donnera un enrichissement souhaité à ASAX face à l'implémentation concrète que nous avons vécue durant notre stage.

8.1 Introduction

En général, une variable est une zone de mémoire qui sert à stocker des données.
Chaque variable doit être introduite par une déclaration de variables qui lui associe un identificateur, une étendue , une durée de vie et un type de données.

- *L'étendue (scope)* spécifie la portion de texte dans laquelle l'identificateur d'une variable a une signification;
- la *durée de vie* d'une variable est la tranche de temps de l'exécution du programme pendant laquelle la variable existe;
- le *type de données* d'une variable détermine de quel type est le contenu de la variable.

En RUSSEL, seul deux types existent: les entiers (*integer*) et les chaînes de bytes (*string*).

8.2 Types de variables

Il existe trois types de variables en RUSSEL:

- les variables globales externes
- les variables globales internes
- les variables locales

8.2.1 Variable globale externe

Une variable globale externe doit être déclarée au début d'un module.
L'étendue d'une telle variable s'applique à tous les modules qui forment le programme d'analyse complet. Toutefois, pour être effectivement utilisée dans d'autres modules, la déclaration doit être répétée dans chaque module.
La durée de vie est toute la durée d'exécution du programme d'analyse des données d'audit.

8.2.2 Variable globale interne

Une variable globale interne se comporte exactement comme la variable globale externe, sauf que son étendue se limite au module qui contient sa déclaration. Sa durée de vie correspond à la durée d'exécution du programme d'analyse des données d'audit.

8.2.3 Variable locale

Une variable locale est locale relativement à une règle. Elle doit être déclarée immédiatement après le nom de la règle et sa liste de paramètres.
L'étendue se limite à l'intérieur de la règle dans laquelle se trouve la déclaration.
De plus, sa déclaration supplante toute variable globale (interne ou externe) qui porte le même nom ; par contre, des variables locales portant le même nom, mais se trouvant dans des règles différentes, sont distinctes et n'ont rien en commun.
Sa durée de vie est celle de l'exécution de la règle où elle est déclarée.

8.3 L'assignation

8.3.1 Principe

D'un point de vue sémantique, l'instruction d'assignation sert à donner à la variable une nouvelle valeur spécifiée dans l'expression à droite du symbole :=.

var_name := *r_expr*

Au niveau de l'implémentation, n'ayant que deux types pour le langage RUSSEL, il faut distinguer deux cas:

- l'assignation d'une valeur numérique : la zone mémoire qu'occupe la variable est mise à jour par la nouvelle valeur entière;
- l'assignation d'une chaîne de caractères : le pointeur de la variable sur la chaîne pointe après sur la nouvelle chaîne.

8.3.2 Erreur rencontrée

Dans une implémentation concrète, tout marche bien pour l'assignation d'un entier.

Par contre, pour l'assignation d'une chaîne de caractères il se peut que dans certains cas, on n'atteigne pas le résultat escompté et théoriquement attendu.

En effet, prenons le cas de l'assignation à une variable globale d'une chaîne de caractères et énumérons les différents cas qui peuvent se présenter :

variable globale : = variable globale	OK
variable globale : = littéral	OK, il est à noter que nous entendons par littéral toute valeur concrète (exemple : 'TOTO')
variable globale : = variable locale	Ici, nous avons détecté un problème car il y a uniquement redirection de l'adresse de la variable globale vers l'emplacement en mémoire de la variable locale. Mais la durée de vie de la variable locale est limitée à l'exécution de la règle et la mémoire occupée pour la gestion de la règle est désallouée et réinitialisée avant le passage à la règle suivante. A ce moment-là, le contenu de la variable globale est également perdu car elle pointe sur un emplacement temporaire.

variable globale : = paramètre de règle	Nous sommes en face du même problème car le paramètre d'une règle a la même durée de vie et la même étendue qu'une variable locale. Il y a donc perte de la valeur pour la variable globale à la sortie de la règle.
variable globale : = champ du record courant	Le même problème se pose ici, car il y a de nouveau une référence vers un emplacement mémoire qui a une durée de vie limitée au traitement de l'enregistrement courant et qui est libéré lors du passage à l'enregistrement suivant.
variable globale : = résultat d'une fonction C	Il y a également problème car la durée de vie de l'emplacement mémoire où est placé le résultat de la fonction est limité au traitement de l'enregistrement courant.
assignation à une variable globale à l'intérieur d'une fonction C pour laquelle cette variable globale est passée comme paramètre par référence	Même problème que précédemment, cette zone mémoire est libérée au passage à l'enregistrement suivant.

Fig. 8.1 : Tableau des assignations possibles d'une chaîne de caractères à une variable globale

La solution que nous proposons aux cinq problèmes expliqués dans le tableau précédent est commune. Pour l'assignation à une variable globale de la valeur d'une variable locale, d'un paramètre, d'un champ de l'enregistrement courant, du résultat d'une fonction C ou d'une valeur à l'intérieur d'une fonction C (où la variable globale est passée comme paramètre par référence), nous proposons la création d'une zone mémoire pour la variable globale lors de sa déclaration. Cette zone mémoire contient par défaut la chaîne vide. Rappelons ici qu'une chaîne de caractères est représentée en RUSSEL par sa longueur et la chaîne proprement dite ; donc, la chaîne vide sera représentée par la valeur 0.
La zone allouée à la déclaration est réallouée dynamiquement à chaque assignation pour permettre le copiage
de la chaîne de caractères. Cette réallocation se fait à l'aide de la fonction *realloc* du langage C.

8.4 Enrichissement possible d'ASAX

Durant notre stage, au moment de l'implémentation de routines C pour l'extension des tables associatives, nous avons rencontré un problème pour la gestion des erreurs de celles-ci.

Au début, nous gérions cela de la manière suivante : dans la routine C, au moment de l'erreur, et après l'affichage de celle-ci, nous sortions de la routine par l'instruction *return* habituelle mais avec un code d'erreur négatif (i.e. que si une procédure finissait normalement, le code retourné était 0 ou positif ; une procédure avec erreur retournait un code négatif (par rapport à l'erreur produite)). Dans le programme ASAX, l'analyse de ce code d'erreur à chaque fois que nous avions utilisé les tables associatives nous permettait de déterminer si l'analyse pouvait continuer ou non. Il est aisé de comprendre que ce type de gestion était très lourd.

Ensuite, nous avons décidé d'utiliser l'instruction *exit* dans les routines C pour quitter non seulement cette routine mais également le programme d'analyse ASAX.

Vu l'expérience que nous avons acquise dans la création de routines C définies par l'utilisateur, nous proposons une toute petite évolution possible à ASAX et qui permettrait une plus grande facilité à la gestion de ce type de routines C. Il suffirait d'ajouter une petite instruction du style **break** qui permettrait d'arrêter l'analyse en cours.

Prenons un exemple, si, dans nos tables associatives, nous avons besoin d'allouer de la mémoire mais qu'il n'y en a plus, cela ne sert à rien de continuer l'exécution du programme d'analyse. Face à la non-disponibilité de la mémoire, la routine C exécutée renvoie un résultat négatif exprimant le manque de mémoire. Lors de l'analyse du résultat de la routine C, le programme ASAX réalisera qu'il peut arrêter son exécution car elle est devenue inutile. C'est là que l'instruction **break** interviendrait.

Lors du retour d'une routine C, on aurait donc une règle du style :

```
rule name ;
var c : integer ;
begin  .....
       c := c_routine(...) ;
       if c<0 → break fi ;
       .....
end ;
```

9. Tests et exploitation des résultats

Dans ce chapitre, à partir d'une collection de virus les plus répandus dans le monde, nous allons voir le taux de détection de notre programme réalisé à partir des règles spécifiques développées dans le chapitre 6. Ensuite, nous montrerons des cas d'exécutions spécifiques de virus face à ces différentes règles.

9.1 Tests

9.1.1 Environnement de test

Pour effectuer l'exécution du programme d'analyse sur les différents virus, nous avions à notre disposition la configuration suivante :

- Intel 486 DX-33,
- 16 MB RAM,
- système d'exploitation Linux.

9.1.2 Déroulement

Durant notre stage à Hambourg, nous avons eu accès à 215 virus issus de la large collection de virus dont le VTC (*Virus Test Center*) dispose actuellement. La collection du centre est énorme, elle contient des virus provenant du monde entier. Une grande partie de ces virus proviennent de laboratoires de recherche ou d'utilisateurs ou entreprises ayant subi une infection mais qui ont découvert le virus avant sa propagation hors du système. Par contre, s'ils ont connu une large diffusion (et parfois médiatique), ces virus sont appelés *wild viruses*, ce sont les virus que l'on retrouve chez les utilisateurs, dans les entreprises et dans les réseaux et qui font des dégâts. Ces virus sont donc les plus importants à détecter.

Il est à rappeler qu'une première règle générique (infection de fichiers exécutables) avait déjà été implémentée partiellement par Morton Swimmer. Après avoir développé totalement cette règle, nous avons analysé manuellement les audit-trails de certains virus non détectés par cette unique règle que nous avions. De cette analyse statique (et très contraignante), ne connaissant alors pratiquement rien aux virus, nous avons pu avoir une idée d'autres types d'attaques possibles. Dès lors, nous avons pu développer d'autres règles génériques et les intégrer avec la première.

Certains de ces virus provoquaient l'arrêt de l'émulateur 8086. En effet, il arrivait que certains virus utilisent des instructions machine non implémentées dans l'émulateur, il y avait alors une erreur de l'émulateur qui rendait impossible l'analyse de l'audit-trail conséquent à cette exécution partielle. Le centre est d'ailleurs à la recherche d'un émulateur plus performant que celui dont il dispose actuellement.

Sur les 215 virus dont nous disposions et qui pouvaient tourner dans l'émulateur (i.e. plus ou moins 200), notre programme d'analyse dynamique résultant des règles génériques développées conceptuellement a réussi à détecter la présence d'un virus dans tous les cas. Le taux de détection pour les virus *wild* auxquels nous avons eu accès est donc de 100 %. De

plus, nous avions intégré certaines autres règles pour permettre la recherche de caractéristiques de ces virus. Ces règles ont été développées dans le chapitre 6.

9.2 Exploitation des résultats

Nous allons montrer ici quelques résultats provenant de l'exécution du programme d'analyse ASAX sur l'audit-trail issu de l'exécution de virus sur l'émulateur. Tous les virus que nous allons utiliser pour ces résultats sont évidemment reconnus par différents anti-virus. Nous avons comparé nos résultats à ceux fournis par l'anti-virus F-PROT de la société *Data Fellows Ltd.* Cet anti-virus nous a permis de connaître le vrai nom du virus, i.e. celui utilisé dans la communauté informatique pour le reconnaître.

Les résultats du programme d'analyse se composent de deux parties :

- l'analyse dynamique en elle-même : elle contient l'analyse de l'exécution du virus sous émulation et peut renvoyer trois types de messages à l'utilisateur :
  - ? ? ? : message d'information ;
  - AAA : détection d'une opération suspecte mais ne permettant pas d'affirmer une infection par un virus ;
  - ! ! ! : message résultant de la détection de l'infection d'un virus et résumant le schéma d'attaque.
- l'analyse statistique : cette analyse, lancée après l'analyse complète de l'audit-trail, permet de comparer les fichiers du système avant et après exécution du virus. On peut ainsi trouver des attributs du virus (taille, changement de date, ...).

Pour construire le fichier audit-trail, nous avons réalisé un fichier batch exécuté sous l'émulation. Ce fichier comprend, dans l'ordre, le lancement :

- du programme SEEK expliqué dans la section 7.2.2.2. ;
- du fichier SAMPLE dans lequel est inclus le corps du virus ;
- d'un ensemble de fichiers COM et EXE représentatifs par leur taille (C10.COM, C100.COM, C1000.COM, C10000.COM, E600.EXE, E1000.EXE, E10000.EXE, E50000.EXE). Dans leur nom, on peut trouver le type du fichier (C = COM, E = EXE) et la taille du fichier. Ces différents fichiers ont été créés pour savoir si un virus s'attaquait à des fichiers de toutes tailles, s'il s'attaquait aux fichiers en rajoutant son corps au fichier ou bien en écrasant, ...
- de REBOOT qui termine l'émulation PC-DOS.

9.2.1 Frodo (frodo.com)

9.2.1.1 Résultats

```
begin parsing description file ...
        asax :      analysis.asa
        asax :      report1.asa
Processing audit trail ...
AAA : CS = 9EA0 : A:\COMMAND.COM 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\COMMAND.COM closed after Write BOF
!!! : CS = 0DF9 : A:\COMMAND.COM closed after Write BOF
AAA : CS = 9EA0 : A:\C10.COM 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\C10.COM closed after Write BOF
??? : CS = 0DF9 : A:\C10.COM :  Executed when INFECTED
AAA : CS = 9EA0 : A:\C100.COM 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\C100.COM closed after Write BOF
??? : CS = 0DF9 : A:\C100.COM :  Executed when INFECTED
AAA : CS = 9EA0 : A:\C1000.COM 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\C1000.COM closed after Write BOF
??? : CS = 0DF9 : A:\C1000.COM :  Executed when INFECTED
AAA : CS = 9EA0 : A:\C10000.COM 28 byte(s) written at BOF
```

```
!!! : CS = 9EA0 : A:\C10000.COM closed after Write BOF
??? : CS = 0DF9 : A:\C10000.COM :  Executed when INFECTED
AAA : CS = 9EA0 : A:\E1000.EXE 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\E1000.EXE closed after Write BOF
??? : CS = 0DF9 : A:\E1000.EXE :  Executed when INFECTED
AAA : CS = 9EA0 : A:\E10000.EXE 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\E10000.EXE closed after Write BOF
??? : CS = 0DF9 : A:\E10000.EXE :  Executed when INFECTED
AAA : CS = 9EA0 : A:\E50000.EXE 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\E50000.EXE closed after Write BOF
??? : CS = 0DF9 : A:\E50000.EXE :  Executed when INFECTED
AAA : CS = 9EA0 : A:\E600.EXE 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\E600.EXE closed after Write BOF
??? : CS = 0DF9 : A:\E600.EXE :  Executed when INFECTED
AAA : CS = 9EA0 : A:\REBOOT.COM 28 byte(s) written at BOF
!!! : CS = 9EA0 : A:\REBOOT.COM closed after Write BOF

end of audit trail reached.
Processing completion rules ...

A:\COMMAND.COM has changed in size by 4096 bytes to this new size : 54127
A:\COMMAND.COM has changed date from 30-01-1992 to 30-01-2092
**************************
A:\C10.COM has changed in size by 4096 bytes to this new size : 4106
A:\C10.COM has changed date from 08-06-1993 to 08-06-2093
**************************
A:\C100.COM has changed in size by 4096 bytes to this new size : 4196
A:\C100.COM has changed date from 08-06-1993 to 08-06-2093
**************************
A:\C1000.COM has changed in size by 4096 bytes to this new size : 5096
A:\C1000.COM has changed date from 08-06-1993 to 08-06-2093
**************************
A:\C10000.COM has changed in size by 4096 bytes to this new size : 14096
A:\C10000.COM has changed date from 08-06-1993 to 08-06-2093
**************************
A:\E1000.EXE has changed in size by 4096 bytes to this new size : 5096
A:\E1000.EXE has changed date from 08-06-1993 to 08-06-2093
**************************
A:\E10000.EXE has changed in size by 4096 bytes to this new size : 14096
A:\E10000.EXE has changed date from 08-06-1993 to 08-06-2093
**************************
A:\E50000.EXE has changed in size by 4096 bytes to this new size : 54096
A:\E50000.EXE has changed date from 08-06-1993 to 08-06-2093
**************************
A:\E600.EXE has changed in size by 4096 bytes to this new size : 4696
A:\E600.EXE has changed date from 08-06-1993 to 08-06-2093
**************************
A:\REBOOT.COM has changed in size by 4096 bytes to this new size : 4220
A:\REBOOT.COM has changed date from 11-06-1995 to 11-06-2095
**************************

There were 11 possible virus infection(s) found
There were 10 suspicious action(s) found

The virus attacked COM, EXE files in this test.

end of emulation.

user time div HZ : 8.500000
system time div HZ: 0.510000
total time div HZ : 9.010000

NADF file size : 375.708 bytes
```

9.2.1.2 Analyse

F-PROT nous dit à propos de ce virus :

- Name: Frodo
- Alias: 4096,IDF
- Origin: Israel
- Size: 4096
- Type: Resident Stealth COM/EXE-files

Grâce à l'analyse dynamique, on peut voir que le virus *frodo* s'attaque d'abord aux fichiers COM en commençant par COMMAND.COM (i.e. l'interpréteur de commandes lancé à chaque

session) puis par les autres fichiers COM du système. Il y écrit 28 bytes au début du fichier puis ce fichier est fermé. D'après la règle générique d'infection de fichiers exécutables, il y a donc un virus et le message « *closed after write BOF* » est envoyé à l'utilisateur.
Il est clair que tout n'est pas exprimé dans ce message, il se peut que le virus installe son corps à la fin du fichier mais cela n'est pas exprimé dans l'analyse de l'exécution car ce n'est pas déterminant pour une règle générique. Les messages qui apparaissent pour l'utilisateur ne sont que ceux qui peuvent lui être utiles pour une plus grande connaissance du virus.
Ensuite, le virus infecte les fichiers EXE de la même manière.
L'analyse statistique nous montre dans ce cas différentes caractéristiques du virus :

- on peut voir que les fichiers ont tous augmenté en taille d'une valeur commune de 4096 bytes ; on peut supposer le virus est non réinscripteur, mais ce n'est pas sûr.
- la date des fichiers infectés a changé : en effet, l'année a augmenté d'un siècle. C'est un des stratagèmes utilisés par les virus pour reconnaître qu'ils ont déjà infectés un fichier sans que l'utilisateur ne s'en aperçoive. En effet, quand on demande le contenu d'un répertoire, la date du fichier ne reprend que la décennie et pas le siècle.

9.2.2 Flip (flip2343.com)

9.2.2.1 Résultats

```
begin parsing description file ...
        asax :      analysis.asa
        asax :      report1.asa
Processing audit trail ...
AAA : CS = 9F4D : A:\C100.COM 3 byte(s) written at BOF
??? : CS = 9F4D : A:\C100.COM :  Abnormal time setting to 02:00:62
!!! : CS = 9F4D : A:\C100.COM closed after Write BOF
??? : CS = 9F4D : A:\C100.COM :  Executed when INFECTED
??? : CS = 0DF9 : A:\C100.COM :  Executed when INFECTED
........
AAA : CS = 9F4D : A:\E1000.EXE 28 byte(s) written at BOF
??? : CS = 9F4D : A:\E1000.EXE :  Abnormal time setting to 02:00:62
!!! : CS = 9F4D : A:\E1000.EXE closed after Write BOF
??? : CS = 9F4D : A:\E1000.EXE :  Executed when INFECTED
??? : CS = 0DF9 : A:\E1000.EXE :  Executed when INFECTED
........
AAA : CS = 9F4D : A:\REBOOT.COM 3 byte(s) written at BOF
??? : CS = 9F4D : A:\REBOOT.COM :  Abnormal time setting to 18:10:62
!!! : CS = 9F4D : A:\REBOOT.COM closed after Write BOF

end of audit trail reached.
Processing completion rules ...

A:\C100.COM has changed in size by 2343 bytes to this new size : 2443
A:\C100.COM has changed time from 02:00:00 to 02:00:62
**************************
.............
A:\E10000.EXE has changed in size by 2343 bytes to this new size : 12343
A:\E10000.EXE has changed time from 02:00:00 to 02:00:62
**************************
.............
A:\REBOOT.COM has changed in size by 2343 bytes to this new size : 2467
A:\REBOOT.COM has changed time from 18:10:34 to 18:10:62
**************************
A:\SAMPLE.COM has changed in size by 2343 bytes to this new size : 5686
A:\SAMPLE.COM has changed time from 00:01:46 to 00:01:62
**************************

There were 8 possible virus infection(s) found
There were 8 suspicious action(s) found

The virus attacked COM, EXE files in this test.

end of emulation.

user time div HZ : 5.620000
system time div HZ: 0.280000
total time div HZ : 5.900000

NADF file size : 205.004 bytes
```

9.2.2.2 Analyse

Voici les résultats de F-PROT :

- Name: Flip
- Size: 2343
- Type: Resident COM/EXE-files MBR

Ce virus attaque les fichiers lancés durant l'émulation (qu'ils soient d'extension COM ou EXE) en écrivant 3 bytes en début de fichier. Ensuite, il change l'heure du fichier en mettant les secondes à 62. Voici encore l'illustration d'un stratagème de reconnaissance d'infection pour le virus.
Il est à remarquer que le virus, après avoir infecté le fichier, exécute celui-ci puis ce fichier est de nouveau exécuté. Ce fait exprime que le DOS reprend la main pour l'exécution du fichier, fait qui se remarque par le changement de CS (*codesegment*).
Dans l'analyse statistique, on s'aperçoit que les différents fichiers infectés ont augmenté d'une taille de 2343 bytes. C'est sûrement un virus non réinscripteur.

9.2.3 Hllo (hllonova.com)

9.2.3.1 Résultats

```
begin parsing description file ...
         asax :     analysis.asa
         asax :     report1.asa
Processing audit trail ...
AAA : CS = 105C : A:\SAMPLE.EXE 12300 byte(s) written at BOF
!!! : CS = 105C : A:\SAMPLE.EXE closed after Write BOF
AAA : CS = 105C : A:\MEM.EXE 12288 byte(s) written at BOF
!!! : CS = 105C : A:\MEM.EXE closed after Write BOF
AAA : CS = 105C : A:\CHKDSK.EXE 12288 byte(s) written at BOF
!!! : CS = 105C : A:\CHKDSK.EXE closed after Write BOF
AAA : CS = 105C : A:\FDISK.EXE 12288 byte(s) written at BOF
!!! : CS = 105C : A:\FDISK.EXE closed after Write BOF

end of audit trail reached.
Processing completion rules ...

A:\CHKDSK.EXE has changed in size by -4695 bytes to this new size : 12289
**************************
A:\FDISK.EXE has changed in size by -46119 bytes to this new size : 12289
**************************
A:\MEM.EXE has changed in size by -27785 bytes to this new size : 12289
**************************
A:\SAMPLE.EXE has changed in size by -7700 bytes to this new size : 12300
**************************

There were 4 possible virus infection(s) found
There were 4 suspicious action(s) found


The virus attacked EXE files in this test.

end of emulation.

user time div HZ : 3.070000
system time div HZ: 0.120000
total time div HZ : 3.190000

NADF file size : 120.084 bytes
```

9.2.3.2 Analyse

Voici les résultats de F-PROT :

- Name: HLLO
- Type: Overwriting
- Variant: HLLO.Novademo.A

- HLLO is a family name - all overwriting viruses written in High Level Languages, such as Pascal, C, C++ or Basic, have been grouped under this name.

D'après l'analyse, on peut voir que le virus attaque quelques fichiers d'extension EXE et qu'il y écrit de 12288 à 12300 bytes (F-PROT dit que le virus en écrit 12288).
Avec l'analyse statistique, on peut se rendre compte que le virus est réinscripteur puisque la nouvelle taille du fichier est celle de l'écriture effectuée lors de l'infection. Il n'y a pas de changement de date ni d'heure.

9.2.4 Little_Brother (littb307.com)

9.2.4.1 Résultats

```
begin parsing description file ...
        asax :      analysis.asa
        asax :      report1.asa
Processing audit trail ...
!!! : CS = 0044 : A:\E1000.COM created as Companion file for A:\E1000.EXE
AAA : CS = 0044 : A:\E1000.COM truncated
!!! : CS = 0044 : A:\E1000.COM created as Companion file for A:\E1000.EXE
!!! : CS = 0044 : A:\E10000.COM created as Companion file for A:\E10000.EXE
AAA : CS = 0044 : A:\E10000.COM truncated
!!! : CS = 0044 : A:\E10000.COM created as Companion file for A:\E10000.EXE
!!! : CS = 0044 : A:\E50000.COM created as Companion file for A:\E50000.EXE
AAA : CS = 0044 : A:\E50000.COM truncated
!!! : CS = 0044 : A:\E50000.COM created as Companion file for A:\E50000.EXE
!!! : CS = 0044 : A:\E600.COM created as Companion file for A:\E600.EXE
AAA : CS = 0044 : A:\E600.COM truncated
!!! : CS = 0044 : A:\E600.COM created as Companion file for A:\E600.EXE

end of audit trail reached.
Processing completion rules ...

There were 8 possible virus infection(s) found
There were 4 suspicious action(s) found

The virus attacked COM files in this test.

end of emulation.

user time div HZ : 3.030000
system time div HZ: 0.140000
total time div HZ : 3.170000

NADF file size : 98.380 bytes
```

9.2.4.2 Analyse

Voici les résultats fournis par F-PROT :

- Name: Little Brother
- Size: 299 and 300
- Type: Companion

D'après l'analyse dynamique, on peut voir que le virus crée un fichier compagnon pour un fichier EXE. Ce virus est l'illustration de la règle générique sur la création de fichiers exécutables.
Le corps du virus se met dans le fichier compagnon, celui-ci est assuré alors d'être exécuté le premier (à cause de l'ordre des priorités dans les extensions DOS). On peut également voir que le virus infecte deux fois le même fichier.
L'analyse statistique ne nous donne aucun résultat. Il faut dire que le virus n'a pas besoin de vérifier un changement de date ou d'heure ou tout autre attribut pour savoir s'il a infecté un fichier ; en effet, il n'a qu'à vérifier que pour un fichier EXE, il existe un fichier COM correspondant, et il sait qu'il a déjà infecté ce fichier.

9.2.5 Necros (necros.com)

9.2.5.1 Résultats

```
begin parsing description file ...
        asax :      as1.asa
        asax :      report1.asa
Processing audit trail ...
??? : CS = 0F47 : A:\C10.COM :  Abnormal time setting to 02:00:62
!!! : CS = 0F47 : A:\C10.COM closed after Write BOF
??? : CS = 0F47 : A:\C10.COM :  Executed when INFECTED
!!! : CS = 0F47 : A:\C10.COM closed after Write BOF
??? : CS = 0F47 : A:\C10.COM :  Abnormal time setting to 02:00:62
??? : CS = 0DF9 : A:\C10.COM :  Executed when INFECTED
??? : CS = 0F47 : A:\C100.COM :  Abnormal time setting to 02:00:62
??? : CS = 0F47 : A:\C100.COM :  Abnormal time setting to 02:00:62
??? : CS = 0F47 : A:\C1000.COM :  Abnormal time setting to 02:00:62
??? : CS = 0F47 : A:\C1000.COM :  Abnormal time setting to 02:00:62
??? : CS = 0F47 : A:\C10000.COM :  Abnormal time setting to 02:00:62
??? : CS = 0F47 : A:\C10000.COM :  Abnormal time setting to 02:00:62
!!! : CS = 0F47 : A:\E1000.COM created as Companion file for A:\E1000.EXE
!!! : CS = 0F47 : A:\E10000.COM created as Companion file for A:\E10000.EXE
!!! : CS = 0F47 : A:\E50000.COM created as Companion file for A:\E50000.EXE
!!! : CS = 0F47 : A:\E600.COM created as Companion file for A:\E600.EXE
??? : CS = 0F47 : A:\REBOOT.COM :  Abnormal time setting to 18:10:62

end of audit trail reached.
Processing completion rules ...

A:\C10.COM has changed in size by 1383366788 bytes to this new size : 1383366798
A:\C10.COM has changed time from 02:00:00 to 02:00:62
**************************
A:\C100.COM has changed in size by 1384546346 bytes to this new size : 1384546446
A:\C100.COM has changed time from 02:00:00 to 02:00:62
**************************
A:\C1000.COM has changed in size by 1384545446 bytes to this new size : 1384546446
A:\C1000.COM has changed time from 02:00:00 to 02:00:62
**************************
A:\C10000.COM has changed in size by 1384520573 bytes to this new size : 1384530573
A:\C10000.COM has changed time from 02:00:00 to 02:00:62
**************************
A:\REBOOT.COM has changed in size by 1385759760 bytes to this new size : 1385759884
A:\REBOOT.COM has changed time from 18:10:34 to 18:10:62
**************************

There were 6 possible virus infection(s) found
No suspicious action found

The virus attacked COM files in this test.

end of emulation.

user time div HZ : 3.440000
system time div HZ: 0.250000
total time div HZ : 3.690000

NADF file size : 137.488 bytes
```

9.2.5.2 Analyse

F-PROT nous fournit ces résultats :

- Name: Necros
- Size: 1164
- Type: Resident COM-files
- A polymorphic virus

On peut voir dans l'analyse dynamique, que le virus attaque les fichiers BAT et COM en écrivant en début de fichier puis en changeant l'heure du fichier (passage à 62 secondes). Il attaque également les EXE en leur créant un fichier compagnon. Ce virus est l'exemple des virus qui adoptent un style d'attaque différent pour les différents types de fichiers existants.
On peut remarquer que F-PROT nous dit que c'est un virus polymorphique. C'est une caractéristique que notre programme ne saura jamais analyser parce que le virus peut changer

mille fois de formes, mais lors de son exécution, il attaquera de la même manière. Ce type de virus sera donc toujours détecté.
Dans l'analyse statistique, on peut voir que le virus a changé la taille du fichier. Malheureusement, nous ne savons pas si c'est l'émulateur corrompu qui renvoie cette valeur ou bien si le virus change réellement la taille du fichier à cette valeur.

9.2.6 HII.4629 (ceib4629.com)

9.2.6.1 Résultats

```
begin parsing description file ...
         asax :      analysis.asa
         asax :      report1.asa
Processing audit trail ...
!!! : CS = 1012 : A:\XCOPY.EXE renamed after unlink by A:\SV11
!!! : CS = 1012 : A:\SAMPLE.COM renamed after unlink by A:\SV11
??? : CS = 1008 : A:\SAMPLE.COM :  Executed when INFECTED
!!! : CS = 1012 : A:\SAMPLE.COM renamed after unlink by A:\SV11
??? : CS = 0DD4 : A:\SAMPLE.COM :  Executed when INFECTED
AAA : CS = 1012 : A:\C.EIB created after unlink
..........
end of audit trail reached.
Processing completion rules ...

A:\XCOPY.EXE has changed in size by 4630 bytes to this new size : 16413
A:\XCOPY.EXE has changed time from 12:00:00 to 12:00:58
*************************
.............

There were 36 possible virus infection(s) found
There were 11 suspicious action(s) found

The virus attacked COM, EXE files in this test.

end of emulation.

user time div HZ : 7.180000
system time div HZ: 0.330000
total time div HZ : 7.510000

NADF file size : 405.156 bytes
```

9.2.6.2 Analyse

Ce virus n'a pas été analysé par F-PROT mais est détecté.
Ce virus est l'illustration de la troisième règle générique sur le renommage de fichiers.
On peut voir dans l'analyse dynamique qu'il y a création d'un fichier temporaire SV11, puis que le fichier attaqué (dans notre cas, XCOPY.EXE) est effacé puis remplacé par ce fichier SV11. On peut supposer que le virus lit le fichier, crée alors ce fichier intermédiaire pour y placer son corps plus le corps du fichier attaqué puis il le remplace après l'avoir effacé.
Dans l'analyse statistique, on se rend compte que le fichier a changé de taille (4630 bytes) et que l'heure du fichier a également changé (les secondes sont passées à 58).

10. Conclusion

Depuis une ou deux années, les médias nous parlent de plus en plus de virus informatiques. Quand le très célèbre virus Michelangelo est apparu pour la première fois, tous les journaux se sont rués sur l'affaire. Depuis, tous les utilisateurs d'ordinateurs connaissent l'existence des ces petits parasites. Malheureusement, le mot « virus » provoque maintenant l'hystérie générale.

Quand, en deuxième licence, nous nous sommes retrouvés devant ce sujet de mémoire, nous ne nous attendions pas à être en face d'un sujet aussi ample. Nous connaissions en réalité bien peu de choses sur les virus. Nous avons essayé dans la première partie de ce mémoire d'être aussi précis que possible pour donner un aperçu de ce vaste monde des virus informatiques et des techniques pour les contre-attaquer.

Face à la prolifération actuelle des virus, il est peut-être temps de prendre un autre point de vue pour se défendre de ces programmes malicieux. Avec les outils de mutation, il est possible de créer des centaines de virus à partir d'un virus de départ. Pour contrer ces nouveaux virus, il faut constamment mettre à jour les outils de détection, et en particulier les scanners qui sont le plus généralement utilisés. On se rend compte actuellement qu'une caractéristique des virus change très rarement, c'est leur stratégie d'infection. Dès lors, il faudrait peut-être passer à des outils d'analyse dynamique qui permettent de se rendre compte de la présence d'un virus durant son exécution.

Pour notre stage, nous avons utilisé l'outil ASAX, présenté auparavant, permettant l'analyse dynamique de traces d'exécution de virus pour les détecter. Ces traces étaient réalisées via une émulation de PC-DOS tournant sur une machine UNIX où l'on y réalisait un auditing de la session DOS dans laquelle se déroulait l'exécution du virus en particulier. Le concept d'auditing a également été présenté précédemment. Mais il ne faut pas croire qu'ASAX n'est limité qu'à une analyse *off-line*, i.e. à une analyse de fichiers audit-trail déjà produits avant leur analyse. En effet, il est possible d'utiliser ASAX de manière *on-line*. Prenons le cas d'un administrateur de réseaux qui veut vérifier la présence de virus sur ses stations de travail. Il lui suffirait d'exécuter le programme ASAX qui analyserait les données lui arrivant de toutes les stations de travail du réseau et enverrait les messages d'avertissement à l'administrateur au moment même de la reproduction du virus.

Pour créer ce programme d'analyse, nous avons présenté deux approches conceptuelles possibles pour représenter les schémas d'attaque des virus et les méthodes correspondantes pour traduire ces schémas dans le programme.

Actuellement, le programme développé au stage permet d'automatiser la mise à jour de la base de données dont s'occupe le VTC (*Virus Test Center*) de Hamburg. Face au nombre grandissant de fichiers suspects qu'ils reçoivent chaque semaine, ce programme est le bienvenu pour déterminer la présence d'un virus et rechercher les caractéristiques de celui-ci.

Une évolution possible au programme serait d'analyser plus de fonctions qu'actuellement. En effet, pour le moment, notre programme n'analyse que les fonctions provenant de l'interruption 0x21. Il existe beaucoup d'autres instructions mais PANDORA n'audite qu'une petite centaine de fonctions de l'interruption 21 (alors qu'il y en a plus ou moins 300). Il serait donc intéressant d'étendre les possibilités de PANDORA pour qu'il audite plus de fonctions.

Il serait également intéressant que PANDORA soit étendu pour d'autres interruptions. Par exemple, grâce au traçage de l'interruption 13, on pourrait se rendre compte des dégâts physiques causés par le virus. En effet, l'interruption 13 reprend les fonctions de manipulation des périphériques.
Comme déjà dit auparavant, il serait également intéressant de trouver un émulateur plus puissant, pouvant émuler un processeur plus avancé que le 8086. En effet, certains virus récents provoquent une erreur de l'émulateur lors de leur exécution. Ces virus supposant que l'on utilise un processeur plus avancé utilisent des instructions non implémentées dans l'émulateur.

11. Bibliographie

- **Baudouin Le Charlier, Abdelaziz Mounji, Naji Habra, Isabelle Mathieu**
  Preliminary Report on Advanced Security Audit Trail Analysis on uniX, Namur, Belgique, 2 novembre 1993
  ftp.info.fundp.ac.be/pub/users/amo/papers/asaxSpec.ps.Z

- **Baudouin Le Charlier, Abdelaziz Mounji, Naji Habra, Isabelle Mathieu**
  ASAX : Software Architecture and Rule based Language for Universal Audit Trail Analysis
  In Proceedings of the Second European Symposium on Research in Computer Security (ESORICS), Toulouse, France, novembre 1992
  ftp.info.fundp.ac.be/pub/users/amo/papers/esoric92-paper.gz.Z

- **Baudouin Le Charlier, Abdelaziz Mounji, Denis Zampuniéris, Naji Habra**
  Distributed Audit Trail Analysis, Namur, Belgique
  in Proceedings of the ISOC'95 Symposium on Network and Distributed Systems Security, San Diego, California, 16-17 février 1995
  ftp.info.fundp.ac.be/pub/publications/RP/RP-94-007.ps.Z

- **Baudouin Le Charlier, Abdelaziz Mounji, Morton Swimmer**
  Dynamic detection and classification of computer viruses using general behaviour patterns, Namur, Belgique
  in Proceedings of Fifth International Virus Bulletin Conference, Boston, 20-22 septembre 1995
  ftp.info.fundp.ac.be/pub/users/amo/papers/vb95-paper.ps.Z

- **Baudouin Le Charlier, Naji Habra, Abdelaziz Mounji**
  ASAX : Specification of some extensions to the Analyzer, Namur, Belgique, 26 septembre 1994

- **Baudouin Le Charlier, Naji Habra, Abdelaziz Mounji**
  ASAX : Implementation design of the NADF evaluator, Namur, Belgique, 26 septembre 1994

- **Computing Dictionnary**
  URL http://wombat.doc.ic.ac.uk/

- **David Ferbrache**
  A Pathology of Computer Viruses, Springer-Verlag, 1992
  ISBN 3-540-19610-2

- **Fred Cohen**
  Computer Viruses : Theory and Experiments,
  Computers & Security, Volume 6 (1987), 22-35

- **Helen Custer**
  Au coeur de Windows NT, Microsoft Press, 1993

- **Jan Hruska**
  Virus informatiques et systèmes anti-virus, Editions Masson, Paris, 1992

- **Klaus Brunnstein, Simone Fischer-Hübner, Morton Swimmer**
  Concepts of an Expert System for Virus Detection, Hamburg, Germany

- **Matt Bishop**
  An Overview of Computer Viruses in a Research Environment, Dartmouth College, Hanover, Allemagne

- **Michael Tischer**
  PC intern : Systemprogrammierung, Editions Data Becker, Düsseldorf, Allemagne, 1992

- **Morton Swimmer**
  Fortschrittliche Virus-Analyse - Die Benutzung von statischer und dynamischer Programm-Analyse zur Bestimmung von Virus-Charakteristika, Hamburg, Allemagne, 25 mai 1995
  Mémoire de fin d'études

- **National Computer Security Center**
  A Guide to Understanding Audit in Trusted Systems, 28 juillet 1987

- **Patrick R. Gallagher**
  A Guide to Understanding Audit in Trusted Systems
  URL http://bilbo.isu.edu/security/isl/audit.html

- **Siemens Nixdorf Software S.A.**
  ASAX : Advanced Sequential files Analysis, User's Guide, Namur, Belgique, 5 juillet 1993

- *SINIXS V5.22 Security Features Administrator's Guide*

- **Vesselin Bontchev**
  Future Trends in Virus Writing, Hamburg, Allemagne
  ftp.informatik.uni-hamburg.de/pub/virus/texts/viruses/trends.zip

- **Vesselin Bontchev**
  Possible virus attacks against integrity programs and how to prevent them, Hamburg, Allemagne
  ftp.informatik.uni-hamburg.de/pub/virus/texts/viruses/attacks.zip

- **Vesselin Bontchev**
  The Bulgarian and Soviet Virus Factories, Sofia, Bulgarie
  ftp.informatik.uni-hamburg.de/pub/virus/texts/viruses/factory.zip

- **Vincent Haulotte**
  Etude des virus informatiques et utilisation d'un langage d'analyse d'audit-trail pour leur détection, Namur, Belgique, 1993
  Mémoire de fin d'études

- **W.T. Polk, L.E. Bassham**
  A Guide to the selection of Anti-virus Tools and Techniques, National Institute of Standards and Technology, Computer Security Division, 2 décembre 1992
  ftp.informatik.uni-hamburg.de/pub/virus/texts/viruses/avselect.zip

12. Annexes

12.1 Annexe 1 : Listing du programme d'analyse *Analysis.asa*

```
uses report1;

global external t1,   /* identifier of table 1 = file informations */
       t2,            /* identifier of table 2 = file handle + name */
       t3,            /* identifier of table 3 = drive letter + current path */
       t4, count_t4,  /* identifier of table 4 = file informations before virus infection
                      and count_t4 : index of the last record in the table*/
       t5,  /* identifier of table 5 = names of all known files (help for the research for
                                       companion files) */
       counter, /* counter of lines */
       id,                          /* value of the key for installation of tables 4 and 5 */
       current_drive : integer;  /* number of the current drive */
global current_path : string;    /* string of the current path (drive included) */

/* ************************************************** */

rule installTables;

/*     this rule creates all the tables needed;
       explanation follows.
*/

var c : integer;
begin
   t1 := create_TA(1,0,0,0,0,0,0,1,1,0,1,0,0);
       /* key(1)= filename,(2) = unlink<F=0,T=1>,(3)= rename <T/F>,(4)= writeBOF <T/F>,
          (5)= filelength,(6)= sure length? <T/F>,(7)= attributes,
          (8)= filedate, (9)= filetime, (10)= lastfileposition,
          (11) = name of the old file who was replaced     (only when rename)
          (12) = file newly created or not (boolean)
          (13) = file infected or not
       */
   t2 := create_TA(0,1);      /* key(1)=file handle,(2)=file name */
   t3 := create_TA(0,1);      /* key(1)=drive number, (2)= current path */
   t4 := create_TA(0,1,0,1,1);
   count_t4 := 0;
       /* key(1) = file number (not relevant), (2) = file name,
          (3) = original file size, (4) = original file date, (5) = original file time */
   t5 := create_TA(1,0);
   trigger off for_current recordAnalysis
end;
/* ************************************************** */

rule fill_t45;
/*     this rule fills the table 4 with all the informations concerning the files
       before virus infection and the table 5 with the names of all known files before
       virus infection.
*/
var c1, c2 , Fn, move, new_lfp, val_ind : integer;
    filename, strtemp : string;
begin
       Fn := strToInt(Function);
       if Fn = 65 and strToInt(ret_CF) = 1
       --> trigger off for_next recordAnalysis;
          true
       --> trigger off for_next fill_t45
       fi;
       if Fn = 45
       --> begin
             if match('^[A-Z]:\\',arg_str1) = 1 /* if filename looks like x:\filename or */
             --> filename := arg_str1;
                match('^\\',arg_str1) = 1         /*                        \filename */
             --> begin
                    strtemp := getSubStr(current_path,1,2);
                    filename := concatStr(strtemp,arg_str1)
                 end;
               true
             --> filename := concatStr(current_path,arg_str1)
             fi;
```

```
            filename := upper(filename);     /* puts all the letters in upper case */
            c1 := isMember(t5,filename);
            if c1 = 0
            --> begin
                   val_ind :=2;
                   c1 := retrieve_1(t5,filename,val_ind,id)
                end;
               c1 = -4
            --> begin
                   count_t4 := count_t4 + 1;
                   id := count_t4;
                   c2 := install_g(t4,id,filename,-1,'','');
                   c2 := install_g(t5,filename,id)
                end
            fi
         end;

        Fn = 73
      --> begin

            c1 := update(t4,id,4,getDate(strToInt(ret_DX)));
            c1 := update(t4,id,5,getTime(strToInt(ret_CX)))
         end;

        Fn = 50 and strToInt(arg_AL) = 2
      --> begin
            move := strToInt(arg_DX) + 65536 * strToInt(arg_CX);
            new_lfp := strToInt(ret_AX) + 65536 * strToInt(ret_DX);
                        /* calculates the new position in the file */
             new_lfp := twosc(new_lfp);
            move := twosc(move);
            c1 := update(t4,id, 3, new_lfp - move)
         end
      fi
end;

/* ************************************************  */
rule recordAnalysis;

/*      this rule detects if the codesegment belongs to the virus
        and triggers off ruleTrigger
*/

var c, Fn, handle, handle_good : integer;
begin

   counter := counter + 1;
/* see if the handle of the file is higher than 4 (because between 1 and 4 : I/O system) */

   handle_good := 0;
   handle := -1 ;
   Fn := strToInt(Function);
   if (Fn = 44 or Fn = 81 or Fn = 45 or Fn = 96)
       --> handle := strToInt(ret_AX);
      (Fn = 47 or Fn = 46 or Fn = 48 or Fn = 50 or Fn = 53 or Fn = 54 or Fn = 73 or Fn = 74)
       --> handle := strToInt(arg_BX)
   fi;
   if handle < 0 or handle > 4
       --> handle_good := 1
   fi;

   if handle_good = 1
       -->
             /* we only consider the functions who have their ret_CF = 0,
                i.e. the function call was successful */

             if ((Fn = 44 or Fn = 81 or Fn = 45 or Fn = 46 or Fn = 47 or Fn = 48
                or Fn = 49 or Fn = 50 or (Fn = 51 and strToInt(arg_AL) = 0) or Fn = 53
                or Fn = 54 or Fn = 59 or Fn = 72 or Fn = 73 or Fn = 74 or Fn = 96)
                and strToInt(ret_CF) = 0)

                    --> begin
                           trigger off for_current ruleTrigger;
                           trigger off for_next recordAnalysis
                        end;

                (Fn = 3 or Fn = 13 or (Fn = 43 and strToInt(ret_CF)=0))

                    --> begin
                           trigger off for_current ruleTrigger;
                           trigger off for_next recordAnalysis
```

```
                        end;

                (Fn = 64 and arg_str1 = 'A:\*.*')

                    --> trigger off for_current fill_t45;

                true
                    --> trigger off for_next recordAnalysis
            fi;
    true
      --> trigger off for_next recordAnalysis
  fi
end;

/* ******************************************************  */

rule ruleTrigger;
var Fn: integer;

/*      this rule triggers the good rule for the updating of the associative tables
        regarding to the operation
*/

begin
   Fn := strToInt(Function);
   if Fn =  3  --> trigger off for_current setDrive;
      Fn = 13  --> trigger off for_current getDefaultDrive;
      Fn = 43  --> trigger off for_current changeDir;
      Fn = 44  --> trigger off for_current createFile;
      Fn = 45  --> trigger off for_current openFile;
      Fn = 46  --> trigger off for_current closeFile;
      Fn = 47  --> trigger off for_current readFile;
      Fn = 48  --> trigger off for_current writeFile;
      Fn = 49  --> trigger off for_current unlinkFile;
      Fn = 50  --> trigger off for_current lseekFile;
      Fn = 51  --> trigger off for_current getSetAttribFile;
      Fn = 53  --> trigger off for_current duplicate;
      Fn = 54  --> trigger off for_current forceDuplicate;
      Fn = 59  --> trigger off for_current execFile;
      Fn = 72  --> trigger off for_current renameFile;
      Fn = 73  --> trigger off for_current getDateTimeFile;
      Fn = 74  --> trigger off for_current setDateTimeFile;
      Fn = 81  --> trigger off for_current createFile;
      Fn = 96  --> trigger off for_current extendedOpenFile
   fi

end;

/* ******************************************************  */

rule setDrive;

/*      updates associative tables for the function 3 : set drive
*/

var c1, c2, drive, val_ind : integer;
    path, drive_letter: string;
begin
       if strToInt(arg_DL) > strToInt(ret_AL) /* if requested drive not present */

            --> skip;

          true

            --> begin
                   current_drive := strToInt(arg_DL);
                   drive_letter := chr(current_drive+65);

       /* installation of the new drive if not existing and construction of the path */

                   c1 := isMember(t3,current_drive);
                   if c1 = -4
                          --> c2 := install_g(t3,current_drive,
                                             concatStr(drive_letter,':'))
                   fi;
                   val_ind := 2;
                   c1 := retrieve_1(t3,current_drive, val_ind,path);
                   current_path := concatStr(path,'\')
                end
       fi
end;
```

```
/* ************************************************** */

rule getDefaultDrive;

/*      updates associative tables for the function 13 : get drive default
*/

var     c1, c2, drive, val_ind : integer;
        path, drive_letter : string;
begin
        current_drive := strToInt(ret_AL);
        drive_letter := chr(current_drive+65);
        c1 := isMember(t3,current_drive);
        if c1 = -4
                --> c2 := install_g(t3,current_drive, concatStr(drive_letter,':'))
        fi;
        val_ind := 2;
        c1 := retrieve_1(t3,current_drive, val_ind, path);
        current_path := concatStr(path,'\')
end;

/* ************************************************** */

rule changeDir;

/*      updates associative tables for the function 43 : change directory
*/

var drive_letter, sub_str, letter, path : string;
    c1, c2, drive_number, long : integer;

begin
        if getSubStr(arg_str1,2,1) = ':'  /* if the DOS command looks like : cd x:... ,
                                             where x is a drive letter */
        --> begin
                drive_letter := getSubStr(arg_str1,1,1);
                drive_number := strToInt(drive_letter) - 65;
                long := strLen(arg_str1);
                letter := getSubStr(arg_str1,long,1);
                if letter = '\'
                        --> path := getSubStr(arg_str1,1,long-1);
                true
                        --> path := arg_str1
                fi;
                c1 := isMember(t3,drive_number);
                if c1 = -4
                        --> c2 := install_g(t3,drive_number,path);
                c1 = 0
                        --> c2 := update(t3,drive_number,2,path)
                fi;
                if drive_number = current_drive
                        --> current_path := concatStr(path,'\')
                fi
            end;

        (arg_str1 = '..') and (strLen(current_path) > 3) /* if the DOS command looks like
                                                            cd .. and path isn't the main
                                                             directory */
        --> begin
                current_path := getSubStr(current_path,1,strLen(current_path)-1);
                do getSubStr(current_path,strLen(current_path),1) != '\'
                        --> current_path := getSubStr(current_path,1,strLen(current_path)-1)
                od;
                drive_letter := getSubStr(current_path,1,1);
                drive_number := strToInt(drive_letter) - 65;
                c2:= update(t3,drive_number,2,getSubStr(current_path,1,strLen(current_path)-1))
            end
        fi
end;

/* ************************************************** */

rule createFile;

/*      updates associative tables for the function 44 or 81 : create
*/

var     c1, c2, handle, val_ind, val_unlink : integer;
        filename, strtemp : string;
```

```
/* if fn = 44 --> file truncated
   if fn = 81 --> fail if file exists
*/

begin

   if match('^[A-Z]:\\',arg_str1) = 1    /* if filename looks like x:\filename or */
     --> filename := arg_str1;
      match('^\\',arg_str1) = 1          /*                              \filename   */
     --> begin
            strtemp := getSubStr(current_path,1,2);
            filename := concatStr(strtemp,arg_str1)
         end;
      true
     --> filename := concatStr(current_path,arg_str1)
   fi;
   filename := upper(filename);     /* puts all the letters in upper case */
   c1 := isMember(t5,filename);
   if c1 = 0
     --> skip;
      c1 = -4
     --> begin
            count_t4 := count_t4 + 1;
            c2 := install_g(t4,count_t4,filename,-1,'','');
            c2 := install_g(t5,filename,count_t4)
         end
   fi;
   handle := strToInt(ret_AX);
   c1 := isMember(t2, handle);
   if c1 = -4
      --> c2 := install_g(t2, handle, filename);  /* install filename */
      c1 = 0
      --> c2 := update(t2, handle, 2, filename)
   fi;

   c1 := isMember(t1,filename);
   if c1 = 0
      --> begin
            val_ind := 2;
            c2 := retrieve_1(t1,filename,val_ind,val_unlink);
            if val_unlink = 1
                  --> begin
                        trigger off for_current postAnalysis;
                        trigger off for_current foundAttempt(CS,filename,
                                                   'created after unlink')

                     end;
               true --> begin
                        trigger off for_current postAnalysis;
                     trigger off for_current foundAttempt(CS,filename,'truncated')
                        end
            fi;
            c2 := Remove(t1,filename)
         end;
      true
      --> trigger off for_current postAnalysis
   fi;
   c2 := install_g(t1,filename,0,0,0,0,0,strToInt(arg_CX),'-','-',1,'',1,0)
end;

/* ************************************************  */

rule openFile;

/*     updates associative tables for the function 45 : open
*/

var c1, c2, handle, val_tab, val_ind : integer;
    val_key, filename, strtemp : string;

begin
   if match('[A-Z]:\\',arg_str1) = 1
     --> filename := arg_str1;
      match('^\\',arg_str1) = 1
     --> begin
            strtemp := getSubStr(current_path,1,2);
            filename := concatStr(strtemp,arg_str1)
         end;
      true
     --> filename := concatStr(current_path,arg_str1)
```

```
   fi;
   c1 := isMember(t5,filename);
   if c1 = 0
       --> skip;
      c1 = -4
       --> begin
              count_t4 := count_t4 + 1;
              c2 := install_g(t4,count_t4,filename,-1,'','');
              c2 := install_g(t5,filename,count_t4)
           end
   fi;

   filename := upper(filename);
   handle := strToInt(ret_AX);
   c1 := isMember(t2, handle);
   if c1 = -4
       --> c2 := install_g(t2, handle, filename);  /* install the filename */
      c1 = 0
       --> c2 := update(t2, handle, 2, filename)
   fi;
   c1 := isMember(t1,filename);
   if c1 = 0
       --> begin
              c2 := update(t1, filename, 10,1);  /* lfp = 0 */
              c2 := update(t1, filename, 4, 0)   /* write BOF false */
           end;
      c1 = -4
       --> c2 := install_g(t1,filename,0,0,0,0,0,-1,'-','-',1,'',0,0)
   fi;
   trigger off for_current postAnalysis
end;

/* ******************************************************  */

rule extendedOpenFile;

/*      updates associative tables for the function 96 : extended open/create
*/

begin
       if strToInt(ret_CX) = 1
              --> trigger off for_current openFile;
          strToInt(ret_CX) = 2 or strToInt(ret_CX) = 3
              --> trigger off for_current createFile
       fi
end;

/* ******************************************************  */

rule closeFile;

/*      updates associative tables for the function 46 : close
*/

var c1, c2, c3, handle, val_tab, val_ind : integer;
    val_key : string;

begin
   handle := strToInt(arg_BX);
   c1 := isMember(t2, handle);

   if c1 = -4
       --> skip;
      c1 = 0
       --> begin
              val_tab := t2;
              val_ind := 2;
              c1 := retrieve_1(val_tab, handle, val_ind, val_key);
              c2 := isMember(t1,val_key);
              if c2 = -4
                     --> c2 := install_g(t1,val_key,0,0,0,0,0,-1,'-','-',0,'',0,0)
              fi;
              trigger off for_current postAnalysis
           end
   fi
 end;

/* ******************************************************  */

rule readFile;
```

```
/*      updates associative tables for the function 47 : read
*/

var c1, c2, handle, val_tab, val_ind,val_lfp : integer;
    bytes_to_read, bytes_read : integer;
    val_key : string;

begin
   handle := strToInt(arg_BX);
   val_tab := t2;
   val_ind := 2;
   c1 := retrieve_1(val_tab, handle, val_ind, val_key);
   bytes_read := strToInt(ret_AX);  /* number of bytes actually read */
   val_tab := t1;
   val_ind := 10;
   c1 := retrieve_1(val_tab, val_key, val_ind, val_lfp);
   c1 := update(val_tab, val_key,10, val_lfp + bytes_read);  /* new  position in the file */
   trigger off for_current postAnalysis
end;

/* ****************************************************  */

rule writeFile;

/*      updates associative tables for the function 48 : write
*/

var c1, c2, handle, val_tab, val_ind, val_write, val_file_length, val_sure_length,val_lfp,
    val_create, index,origin_size : integer;
    val_key : string;

begin
   handle := strToInt(arg_BX);
   val_tab := t2;
   val_ind := 2;
   c1 := retrieve_1(val_tab, handle, val_ind, val_key);
   val_tab := t1;
   val_ind := 5;
   c1 := retrieve_1(val_tab, val_key, val_ind, val_file_length);
   val_ind := 12;
   c1 := retrieve_1(val_tab, val_key, val_ind, val_create);
   val_ind := 6;
   c1 := retrieve_1(val_tab, val_key, val_ind, val_sure_length);
   val_ind := 10;
   c1 := retrieve_1(val_tab, val_key, val_ind, val_lfp);
val_ind := 2;
val_tab := t5;
c1 := retrieve_1(val_tab,val_key,val_ind,index);
val_ind := 3;
val_tab := t4;
c1 := retrieve_1(val_tab,val_key,val_ind,origin_size);

   if ((val_file_length <= val_lfp) or
      ((val_file_length > val_lfp) and (val_file_length < val_lfp + strToInt(ret_AX))))
       and val_sure_length = 1 /* look for the fact that the length is sure */
       --> begin
              c1 := update(t1, val_key, 10, val_lfp + strToInt(ret_AX));
              c1 := update(t1, val_key, 5, val_lfp + strToInt(ret_AX))
                          /* the size of the file has increased */
           end;

val_lfp + strToInt(ret_AX) > origin_size
and val_sure_length = 0
       --> begin
              c1 := update(t1, val_key, 10, val_lfp + strToInt(ret_AX));
              c1 := update(t1, val_key, 5, val_lfp + strToInt(ret_AX));
              c1 := update(t1, val_key, 6, 1)
           end;

      true --> c1 := update(t1, val_key, 10, val_lfp + strToInt(ret_AX))

   fi;

   if ((match('.*COM$',val_key) = 1
      and val_lfp = 0)
     or(match('.*EXE$',val_key) = 1
      and val_lfp < 20))
     and val_create = 0                /* if write attempt at BOF (begin of file) */
       --> c1 := update(t1, val_key, 4, 1);
      val_lfp < 20 and val_create = 0 /*  for all the other types of files:
                                          may be .COM or .EXE */
```

```
        --> c1 := update(t1, val_key, 4, 1)
    fi;
    trigger off for_current postAnalysis
end;

/* ************************************************  */

rule unlinkFile;

/*      updates associative tables for the function 49 : unlink
*/

var c1 : integer;
    filename,strtemp : string;

begin
    if match('[A-Z]:\\',arg_str1) = 1
        --> filename := arg_str1;
       match('^\\',arg_str1) = 1
        --> begin
                strtemp := getSubStr(current_path,1,2);
                filename := concatStr(strtemp,arg_str1)
            end;
       true
        --> filename := concatStr(current_path,arg_str1)
    fi;

    filename := upper(filename);
    c1 := update(t1, filename, 2, 1)
end;

/* ************************************************  */

rule lseekFile;

/*      updates associative tables for the function 50 : lseek
*/

var c1, c2, handle, val_tab, val_ind, new_lfp,new_lfp2, move : integer;
    val_key : string;

begin
    handle := strToInt(arg_BX);
    val_tab := t2;
    val_ind := 2;
    c1 := retrieve_1(val_tab, handle, val_ind, val_key);
    if c1 = 0
    -->  begin
                move := strToInt(arg_DX) + 65536 * strToInt(arg_CX);
                new_lfp := strToInt(ret_AX) + 65536 * strToInt(ret_DX);
                                /* calculates the new position in the file */
                 new_lfp2 := twosc(new_lfp);
                c1 := update(t1, val_key, 10, new_lfp2)
                trigger off for_current postAnalysis
        end
    fi
end;

/* ************************************************  */

rule getSetAttribFile;

/*      updates associative tables for the function 51 : get or set attributes
        of file
*/

var c1, c2 : integer;
    filename ,strtemp: string;
begin
    if match('[A-Z]:\\',arg_str1) = 1
        --> filename := arg_str1;
       match('^\\',arg_str1) = 1
        --> begin
                strtemp := getSubStr(current_path,1,2);
                filename := concatStr(strtemp,arg_str1)
            end;
       true
        --> filename := concatStr(current_path,arg_str1)
    fi;
    c1 := isMember(t5,filename);
    if c1 = 0
```

```
        --> skip;
      c1 = -4
        --> begin
              count_t4 := count_t4 + 1;
              c2 := install_g(t4,count_t4,filename,-1,'','');
              c2 := install_g(t5,filename,count_t4)
            end
   fi;

   filename := upper(filename);
   if strToInt(arg_AL) = 0
       --> c1 := update(t1, filename, 7, strToInt(ret_CL));
      strToInt(arg_AL) = 1
       --> c1 := update(t1, filename, 7, strToInt(arg_CL))
   fi;
   trigger off for_current postAnalysis
end;

/* ************************************************** */

rule duplicate;

/*      updates associative tables for the function 53 : duplciate file handle
*/

var c, val_ind, file_handle, new_handle: integer;
    filename : string;
begin
   file_handle := strToInt(arg_BX);
   val_ind := 2;
   c := retrieve_1(t2, file_handle, val_ind, filename);
   new_handle := strToInt(ret_AX);
   c := isMember(t2, new_handle);
   if c = 0
     --> c := Remove(t2, new_handle)
   fi;
   c := install_g(t2, new_handle, filename)
end;

/* ************************************************** */

rule forceDuplicate;

/*      updates associative tables for the function 54 : force duplicate of file handle
*/

var c, val_ind, file_handle, new_handle: integer;
    filename : string;
begin
   file_handle := strToInt(arg_BX);
   val_ind := 2;
   c := retrieve_1(t2, file_handle, val_ind, filename);
   new_handle := strToInt(arg_CX);
   c := isMember(t2, new_handle);
   if c = 0
     --> c := Remove(t2, new_handle)
   fi;
   c := install_g(t2, new_handle, filename)
end;


/* ************************************************** */

rule execFile;

/*      looks for the fact that the virus try to execute an infected file or not
*/

var c1, c2 : integer;
    filename, strtemp : string;

begin
   if match('[A-Z]:\\',arg_str1) = 1
       --> filename := arg_str1;
      match('^\\',arg_str1) = 1
       --> begin
              strtemp := getSubStr(current_path,1,2);
              filename := concatStr(strtemp,arg_str1)
            end;
      true
       --> filename := concatStr(current_path,arg_str1)
```

```
   fi;
   filename := upper(filename);
   c1 := isMember(t5,filename);
   if c1 = 0
       --> skip;
      c1 = -4
       --> begin
              count_t4 := count_t4 + 1;
              c2 := install_g(t4,count_t4,filename,-1,'','');
              c2 := install_g(t5,filename,count_t4)
           end
   fi;
   c1 := isMember(t1,filename);
   if c1 = 0
       --> skip;
      c1 = -4
       --> c2 := install_g(t1,filename,0,0,0,0,0,-1,'','',0,'',0,0)
   fi;
   trigger off for_current postAnalysis

end;

/* ************************************************  */

rule renameFile;

/*     updates associative tables for the function 72 : rename
*/

var c1, c2,
    vtab, nbunlink, nbrename, nbwrite,vfile_length,
    vsure_length, vattrib, vlfp, val_create, val_infected : integer;
    val_key, vfile_date,vfile_time ,new_name, filename, filename2,strtemp : string;

begin
   if match('[A-Z]:\\',arg_str1) = 1
       --> filename := arg_str1;
      match('^\\',arg_str1) = 1
       --> begin
              strtemp := getSubStr(current_path,1,2);
              filename := concatStr(strtemp,arg_str1)
           end;
      true
       --> filename := concatStr(current_path,arg_str1)
   fi;
   filename := upper(filename);
   c1 := isMember(t5,filename);
   if c1 = 0
       --> skip;
      c1 = -4
       --> begin
              count_t4 := count_t4 + 1;
              c2 := install_g(t4,count_t4,filename,-1,'','');
              c2 := install_g(t5,filename,count_t4)
           end
   fi;
   val_key := filename;   /* research of all the values */
   c1 := retrieve_g(t1, val_key,nbunlink, nbrename, nbwrite,
                    vfile_length, vsure_length, vattrib, vfile_date, vfile_time, vlfp,
                    new_name,val_create,val_infected);
   if match('^[A-Z]:\\',arg_str2) = 1
       --> filename2 := arg_str2;
      match('^\\',arg_str2) = 1
       --> begin
              strtemp := getSubStr(current_path,1,2);
              filename2 := concatStr(strtemp,arg_str2)
           end;
      true
       --> filename2 := concatStr(current_path,arg_str2)
   fi;
   filename2 := upper(filename2);
   c1 := isMember(t5,filename2);
   if c1 = 0
       --> skip;
      c1 = -4
       --> begin
              count_t4 := count_t4 + 1;
              c2 := install_g(t4,count_t4,filename2,-1,'','');
              c2 := install_g(t5,filename2,count_t4)
           end
   fi;
```

```
            /* if the command looks like : rename A B
               if B already exists, A is put as parameter for the file B (parameter 14)
               if B doesn't exist in the table, creation of a new entry with the old values
                  of A but with the name B */
   c1 := isMember(t1, filename2);
   if c1 = 0
       --> begin
              c2 := update(t1, filename2, 11, filename);
              c2 := update(t1, filename2, 3, 1)
           end;
      c1 = -4
       --> begin
              c2 := install_g(t1,filename2,0,1,0,vfile_length,vsure_length,vattrib,
                              vfile_date,vfile_time,0,'',0,val_infected);
              c2 := Remove(t1, filename)
           end
   fi;

   trigger off for_current postAnalysis
end;

/* ******************************************************  */

rule getDateTimeFile;

/*      updates associative tables for the function 73 : get date and time of file
*/

var c1, c2, handle, val_ind : integer;
    val_key : string;

begin
   handle := strToInt(arg_BX);
   val_ind := 2;
   c1 := retrieve_1(t2, handle, val_ind, val_key);
   c1 := update(t1, val_key, 8,getDate(strToInt(ret_DX)));
   c1 := update(t1, val_key, 9,getTime(strToInt(ret_CX)));
   trigger off for_current postAnalysis
end;

/* ******************************************************  */

rule setDateTimeFile;

/*      updates associative tables for the function 74 : set date and time of file
*/

var c1, c2, handle, val_ind : integer;
    val_key : string;

begin
   handle := strToInt(arg_BX);
   val_ind := 2;
   c1 := retrieve_1(t2, handle, val_ind, val_key);
   c1 := update(t1, val_key, 8,getDate(strToInt(arg_DX)));
   c1 := update(t1, val_key, 9,getTime(strToInt(arg_CX)));

   trigger off for_current postAnalysis
end;

/* ******************************************************  */

rule postAnalysis;

/*      This rule analyses the associative tables after modification by the rule
        regarding the operation
*/

var val_unlink, val_rename, val_write, val_file_length,
    val_sure_length, val_attrib, val_lfp, val_create, val_infected,
    Fn, c1, c2, handle, val_ind, long : integer;
    val_key, val_file_rename, val_file_rename2, val_file_date, val_file_time, filename,
    file_temp : string;
begin
   Fn := strToInt(Function);

/* research of the values of the file being considered */
   if Fn = 44 or Fn = 81 or Fn = 59
       --> begin
              if match('[A-Z]:\\',arg_str1) = 1
```

```
                --> filename := arg_str1;
             getSubStr(arg_str1,1,1) = '\'
                --> filename := concatStr(getSubStr(current_path,1,2), arg_str1);
             true
                --> filename := concatStr(current_path,arg_str1)
          fi;
          filename := upper(filename)
       end;
   Fn = 46 or Fn = 48 or Fn = 74
    --> begin
          handle := strToInt(arg_BX);
          val_ind := 2;
          c1 := retrieve_1(t2, handle, val_ind, val_key);
          filename := val_key
       end
fi;

if (Fn = 44 or Fn = 81) /* create */
   --> begin
          if match('.*COM$',filename) = 1
                --> begin
                      long := strLen(filename);
                      file_temp := getSubStr(filename,1,long - 3);
                      file_temp := concatStr(file_temp,'EXE');  /* if creation in a
                                                COM file
                                                and an EXE file with the same name already
                                                exists, it may be a companion virus */
                      c1 := isMember(t5,file_temp);
                      if c1 = 0
                      --> trigger off for_current
          foundVirus(CS,filename,concatStr('created as Companion file for ',file_temp));
                         c1 = -4
                      --> trigger off for_current
                            foundAttempt(CS,filename,'created as executable')
                      fi
                   end;
             match('.*EXE$',filename) = 1 or match('.*BAT',filename) = 1
             --> trigger off for_current
                            foundAttempt(CS,filename,'created as executable')
          fi
      end;

   Fn = 45  /* open */
   --> skip;

  Fn = 46  /* close */
   --> begin
          val_ind := 4;
          c1 := retrieve_1(t1,filename,val_ind,val_write);
          if val_write = 1
                --> begin
                      trigger off for_current
                             foundVirus(CS,filename,'closed after Write BOF');
                      c1 := update(t1,filename,7,2)
                   end
          fi
       end;

   Fn = 47  /* read */
   --> skip ;

   Fn = 48  /* write */
   --> begin
          val_ind := 12;
          c1 := retrieve_1(t1,filename,val_ind,val_create);
          val_ind := 10;
          c1 := retrieve_1(t1,filename,val_ind,val_lfp);
          if (match('.*COM$',filename) = 1
        and val_create = 0)
          --> if (val_lfp - strToInt(ret_AX) = 0)  /* if write at BOF in a COM file */
          --> trigger off for_current
    foundAttempt(CS,filename,concatStr(int2s(strToInt(ret_AX)),' byte(s) written at BOF'))
                fi;
        match('.*EXE$',filename) = 1
        and val_create = 0
          --> if (val_lfp - strToInt(ret_AX) <= 20)  /* if write at BOF in an EXE file */
                --> trigger off for_current
    foundAttempt(CS,filename,concatStr(int2s(strToInt(ret_AX)),' byte(s) written at BOF'))
                fi;
       val_create = 0
          -->if (val_lfp - strToInt(ret_AX) <= 20)     /*if write at BOF for other
```

```
                                                        types of file*/
                --> trigger off for_current
 foundAttempt(CS,filename,concatStr(int2s(strToInt(ret_AX)),' byte(s) written at BOF'))
             fi
    fi
         end;

Fn = 49  /* unlink */
--> skip;

Fn = 50  /* lseek */
--> skip;

Fn = 51  /* getSetAttribFile */
--> skip;

Fn = 59  /* exec */
--> begin
         val_ind := 13;
         c1 := retrieve_1(t1,filename,val_ind,val_infected);
         if val_infected = 1
         -->println('??? : CS = ',str2h(CS),' : ',filename,' :  Executed when INFECTED')
         fi
      end;

Fn = 72  /* rename */
--> begin
         if match('[A-Z]:\\',arg_str2) = 1
                --> filename := arg_str2;
            true
                --> filename := concatStr(current_path,arg_str2)
         fi;
         filename := upper(filename);
         val_key := filename;
         val_ind := 11;
         c1 := retrieve_1(t1,val_key,val_ind,val_file_rename);
         if val_file_rename = ''
         --> if match('.*COM$',val_key) = 1
                --> begin
                       long := strLen(val_key);
                       file_temp := getSubStr(val_key,1,long - 3);
                       file_temp := concatStr(file_temp,'EXE');
                                    /* if renaming in a COM file
                                    and an EXE file with the same name already
                                    exists, it may be a companion virus */
                       c1 := isMember(t5,file_temp);
                       if c1 = 0
                       --> trigger off for_current
                       foundVirus(CS,filename,'renamed into Companion file');
                          c1 = -4
                       --> trigger off for_current
                       foundAttempt(CS,filename,'renamed into COM file')
                       fi
                   end;
               match('.*EXE$',val_key) = 1 or match('.*BAT$',val_key) = 1
                       /* renaming in an EXE or BAT file (that didn't exist before) */
                --> trigger off for_current
                foundAttempt(CS,filename,'renamed into executable file')
             fi;
          true
         --> begin
                val_ind := 2;
                c1 := retrieve_1(t1,val_key,val_ind,val_unlink);
                val_ind := 12;
                c1 := retrieve_1(t1,val_file_rename,val_ind,val_create);
                if val_unlink = 1
                and val_create = 1
                --> begin
                       trigger off for_current
         foundVirus(CS,filename,concatStr('renamed after unlink by ',val_file_rename));
                       c1:= Remove(t1,val_key);
                       if match('[A-Z]:\\',arg_str1) = 1
                       --> filename := arg_str1;
                          true
                       --> filename := concatStr(current_path,arg_str1)
                       fi;
                       filename := upper(filename);
                       if match('[A-Z]:\\',arg_str2) = 1
                       --> file_temp := arg_str2;
                          true
                       --> file_temp := concatStr(current_path,arg_str2)
```

```
                                fi;     /* after having removed the old file, installation
                                           of the new file with the parameters of the old file */
                                file_temp := upper(file_temp);
                                c1 := retrieve_g(t1,filename,val_unlink,val_rename,val_write,
                                             val_file_length, val_sure_length, val_attrib,
                                             val_file_date, val_file_time, val_lfp,
                                             val_file_rename2, val_create,val_infected);
                                c1:= install_g(t1,file_temp,0,0,0,val_file_length,
                                             val_sure_length,val_attrib,val_file_date,
                                             val_file_time,0,'',0,val_infected);
                                c1 := Remove(t1,filename)
                            end
                        fi
                    end
                fi

            end;

        Fn = 73  /* getDateTimeFile */
        --> skip;

        Fn = 74  /* setDateTimeFile */
        --> begin
                val_ind := 8;
                c1 := retrieve_1(t1,filename,val_ind,val_file_date);
                val_ind := 9;
                c1 := retrieve_1(t1,filename,val_ind,val_file_time);
                if  (bytesToInt(getSubStr(val_file_date,1,2)) > 31)
                 or (bytesToInt(getSubStr(val_file_date,4,2)) > 12)
                --> println('??? : CS = ',str2h(CS),' : ',val_key,
                        ' :  Abnormal date setting to ',val_file_date)
                fi;
                if  bytesToInt(getSubStr(val_file_time,1,2)) > 23
                 or bytesToInt(getSubStr(val_file_time,4,2)) > 59
                 or bytesToInt(getSubStr(val_file_time,7,2)) > 59
                --> println('??? : CS = ',str2h(CS),' : ',val_key,
                        ' :  Abnormal time setting to ',val_file_time)
                fi
            end
    fi

end;

/* ************************************************  */

rule statistics;

var i, file_size, file_size2,c1,c2,c3,val_ind,true_length,star  : integer;
    file_name,file_date,file_time, file_date2, file_time2 : string;

/*      this rule analyzes the supposed infected files and checks modifications in size, date
        and time with the files in table 4 (file informations before virus infection)
*/

begin

/*      this part prints the contents of all the tables used
*/

    println('Files :');
    c1 := print_TA(t1);
    println;


    i := 1;
    do
        i <= count_t4
        --> begin
                c1 := retrieve_g(t4,i,file_name,file_size,file_date,file_time);
                c1 := isMember(t1,file_name);
                if c1 = 0
                and file_size != -1
                    --> begin
                        star := 0;
                        val_ind := 6;
                        c2 := retrieve_1(t1,file_name,val_ind,true_length);
                        if true_length = 1
                            --> begin
                                val_ind := 5;
                                c3 := retrieve_1(t1,file_name,val_ind,file_size2);
                                if file_size2 != file_size
```

```
                                --> begin
                                    println(file_name,' has changed in size by ',
                                file_size2 - file_size,' bytes to this new size : ',file_size2);
                                    star := 1
                                    end
                                fi
                                end
                        fi;
                        val_ind := 8;
                        c2 := retrieve_1(t1,file_name,val_ind,file_date2);

                        if file_date2 != '-' and file_date2 != ''
                        and file_date != file_date2
                           --> begin
                                println(file_name,' has changed date from ',file_date, ' to
                                        ',file_date2);
                                star := 1
                                end
                        fi;

                        val_ind := 9;
                        c2 := retrieve_1(t1,file_name,val_ind,file_time2);

                        if file_time2 != '-' and file_time2 != ''
                        and file_time != file_time2
                           --> begin
                                println(file_name,' has changed time from ',file_time, ' to
                                        ',file_time2);
                                star := 1
                                end
                        fi;
                        if star = 1 --> println('**************************') fi
                        end
                fi;
                i := i + 1
            end
    od;


/*      this part frees memory
*/

    c1 := release(t1);
    c1 := release(t2);
    c1 := release(t3);
    c1 := release(t4);
    c1 := release(t5)
end;

/* ************************************************  */

init_action;
begin
    counter := 0;
    trigger off for_next installTables;
    trigger off at_completion statistics
end.
```

12.2 Annexe 2 : Syntaxe du langage RUSSEL

12.2.1 Définition BNF des éléments lexicaux

< token > ::= < identifier >
| < constant >
| < special symbol >

< identifier > ::= < letter >
| < identifier > < digit >
| < identifier > < letter >
| < identifier > < underscore > < letter >
| < identifier > < underscore > < digit >

< constant > ::= < integer constant >
| < C_literal >
| < X_literal >

< integer constant > ::= < digit >
| < integer constant > < digit >

< C literal > ::= ‘ < C_sequence > ‘

< C_sequence > ::= < empty >
| C_sequence > < C_character >

< C_character > ::= any printable ASCII except simple quote (‘) | ‘ ‘

< X literal > ::= X ‘ < X_sequence > ‘

< X sequence > ::= < empty >
| < X_sequence > < X_symbol >

< X_symbol > ::= < X_character > < X_character >

< X_character > ::= 0 | ... | 9 | A | B | C | D | E | F

< digit > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

< letter > ::= a | ... | z | A | ...| Z

< special symbol > ::= + | – | * | (|) | > | < | != | >= | <= | : | ;
| , | %= | = | --> | := | **and** | **at_completion** | **begin** | **div**
| **do** | **end** | **external** | **false** | **fi** | **for_current** | **for_next**
| **global** | **if** | **init_action** | **integer** | **mod** | **internal** |
| **not** | **od** | **off** | **or** | **present** | **rule** | **skip** | **string**
| **trigger** | **true** | **uses** | **var**

12.2.2 Définition BNF de la syntaxe abstraite du langage RUSSEL

< module > ::= < usage list >
< global var decl >
< rule decl list >
< init action > .

< usage list > ::= < empty >
| < module usage > ; ... ; < module usage > ;

< module usage > ::= **uses** < list of modules >

< list of modules > ::= < module name > , ... , < module name >

< module name > ::= identifier

< global var decl > ::= < empty >
| < global var group > ; ... ; < global var group > ;

< global var group > ::= **global** < variable class >
< variable name > , ... , < variable name > : < type >

< variable class > ::= < empty >
| **internal**
| **external**

< rule decl list > ::= < empty >
| < rule declaration > ; ... ; < rule declaration > ;

< rule declaration > ::= < rule heading >
< variable declaration part > < action part >

< rule heading > ::= < rule class >
rule < rule name > (< parameter list >)

< rule class > ::= < empty >
| **internal**
| **external**
< rule name > ::= < identifier >

< parameter list > ::= < empty >
| < parameter group > ; ... ; < parameter group >

< parameter group > ::= < empty >
| < parameter name > , ... , < parameter name >

< parameter name > ::= < identifier >

< type > ::= **string** | **integer**

< variable declaration part > ::= < empty >
| **var** < variable declaration > ; ... ; < variable declaration >

< variable declaration > ::= < variable name > , ... , < variable name > : < type >

< variable name > ::= < identifier >

< action part > ::= < action >

< action > ::= **skip**
| < assignment >
| < conditional action >
| < repetitive action >
| < compound action >
| < rule triggering >
| < predefined procedure call >

< assignment > ::= < left expression > **:=** < right expression >

< left expression > ::= < parameter name >
| < variable name >

< right expression > ::= < expression >

< expression > ::= < constant >
| < field name >
| < left expression >
| < expression > < binary arithmetic operator > < expression >
| < unary arithmetic operator > < expression >

< field name > ::= < identifier >

< conditional action > ::= **if** < guarded action > **;** ... **;** < guarded action > **fi**

< guarded action > ::= < condition > → < action >

< condition > ::= **true**
| **false**
| **present** < field name >
| < expression > < relational operator > < expression >
| < condition > < logical operator > < expression >
| **not** < condition >

< relational operator > ::= < | > | = | != | <= | >=

< logical operator > ::= **and** | **or**

< repetitive action > ::= **do** < guarded action > **;** ... **;** < action > **od**

< coumpound action > ::= **begin** < action > **;** ... **;** < action > **end**

< rule triggering > ::= **trigger off** < triggering mode > < rule call >

< triggering mode > ::= **for_next**
| **for_current**
| **at_completion**

< predefined procedure call > ::= < predefined procedure name >
(< expression > **,** ... **,** < expression > **)**

< predefined procedure name > ::= < identifier >

< rule call > ::= < rule name > **(** < expression > **,** ... **,** < expression > **)**

< rule name > ::= < identifier >

< init action > ::= **init_action ;**
< variable declaration part > < action part >

12.2.3 Définition BNF de la syntaxe concrète du langage RUSSEL

< condition > ::= < disjunction >

< disjunction > ::= < conjunction >
| < disjunction > **or** < conjunction >

< conjunction > ::= < simple condition >
| < conjunction > **and** < simple condition >

< simple condition > ::= **true**
| **false**
| **present** < field name >
| < relational expression >
| **(** < condition > **)**
| **not** < condition >

< relational expresssion > ::= < arithmetic expression > < relational operator >
| < arithmetic expression >

< relational operator > ::= < | > | ≠ | = | ≤ | ≥

< arithmetic expression > ::= < term >
| < arithmetic expression > < additive operator > < term >

< term > ::= < factor >
| < term > < multiplicative operator > < factor >

< factor > ::= < simple expression >

< simple expression > ::= < parameter name >
| < variable name >
| < integer constant >
| **(** < arithmetic expression > **)**
| < pre-defined procedure call >

< multiplicative operator > ::= * | **div** | **mod**

< additive operator > ::= + | –

12.3 Annexe 3 : Le fichier DDF (*Data Description File*) de PANDORA

```
A pandora_1.0
B data description file for PANDORA

1 1
2 word
3 word
4 CS
5 Code Segment

1 2
2 word
3 word
4 IP
5 Instruction Pointer

1 3
2 byte
3 byte
4 RecType
5 Type of record

1 4
2 byte
3 byte
4 Function
5 Normalized function number

1 5
2 dword
3 dword
4 StartTime
5 Function started

1 6
2 dword
3 dword
4 EndTime
5 Function ended

1 7
2 byte
3 byte
4 arg_AL
5 Argument: register AL

1 8
2 word
3 word
4 arg_AX
5 Argument: register AX (Accumulator)

1 9
2 byte
3 byte
4 arg_BL
5 Argument: register BL

1 10
2 word
3 word
4 arg_BX
5 Argument: register BX (Base)

1 11
2 byte
3 byte
4 arg_CL
5 Argument: register CL

1 12
2 word
3 word
4 arg_CX
5 Argument: register CX (Counter)

1 13
2 byte
3 byte
4 arg_DL
5 Argument: register DL

1 14
2 word
3 word
4 arg_DX
5 Argument: register DX (Data)
```

1 15
2 word
3 word
4 arg_SI
5 Argument: register SI (Source Index)

1 16
2 word
3 word
4 arg_DI
5 Argument: register DI (Destination Index)

1 17
2 word
3 word
4 arg_DS
5 Argument: register DS (Data segment)

1 18
2 word
3 word
4 arg_ES
5 Argument: register ES (Extra segment)

1 19
2 byte
3 byte
4 arg_CF
5 Argument: register CF (Carry flag)

1 20
2 string
3 string
4 arg_str1
5 Argument: first string

1 21
2 string
3 string
4 arg_str2
5 Argument: second string

1 22
2 byte
3 byte
4 ret_AL
5 Result : register AL

1 23
2 word
3 word
4 ret_AX
5 Result : register AX

1 24
2 byte
3 byte
4 ret_BL
5 Result : register BL

1 25
2 word
3 word
4 ret_BX
5 Result : register BX

1 26
2 byte
3 byte
4 ret_CL
5 Result : register CL

1 27
2 word
3 word
4 ret_CX
5 Result : register CX

1 28
2 byte
3 byte
4 ret_DL
5 Result : register DL

1 29
2 word
3 word
4 ret_DX
5 Result : register DX

1 30
2 word
3 word
4 ret_SI
5 Result : register SI

1 31
2 word
3 word
4 ret_DI
5 Result : register DI

1 32
2 word
3 word
4 ret_DS
5 DS

1 33
2 word
3 word
4 ret_ES
5 Result : register ES

1 34
2 byte
3 byte
4 ret_CF
5 Result : register CF

1 35
2 string
3 string
4 ret_str1
5 Result : first string

1 36
2 string
3 string
4 ret_str2
5 Result : second string